AFIT/GCS/ENG/00M-21

KNOWLEDGE BASE SUPPORT FOR DESIGN AND SYNTHESIS OF MULTI-AGENT SYSTEMS

THESIS

Marc J Raphael, Captain, USAF

AFIT/GCS/ENG/00M-21

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

AFIT/GCS/ENG/00M-21

KNOWLEDGE BASE SUPPORT FOR DESIGN AND SYNTHESIS OF MULTI-AGENT SYSTEMS

THESIS

Presented to the faculty of the Graduate School of Engineering and Management
of the Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Marc J Raphael, B. S.
Captain, USAF

March 2000

KNOWLEDGE BASE SUPPORT FOR DESIGN AND SYNTHESIS OF MULTI-AGENT SYSTEMS

THESIS

Marc J Raphael, B. S.
Captain, USAF

Approved:

Major Scott A. DeLoach, Committee Chairman	date
Dr. Thomas C. Hartrum, Committee Member	date
Major Robert P. Graham Jr., Committee Member	date

ACKNOWLEDGMENTS

I am at the crossroads with this effort, but I don't know which road brought me the furthest to get here.

-For the thirst for learning, I credit God.

-For the opportunity to learn, I credit my Country. I give her back this small return.

-For pulling me through the tougher days, I credit the support of my wife, Rachel and the smiles, hugs, and laughs of our children, Ethan and Sarah.

-For accepting nothing less than technical excellence and then showing how best to achieve it, I credit Major DeLoach.

Marc J Raphael

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

AgentTool is an AFIT-produced, AFOSR-sponsored multi-agent system (MAS) development tool intended for production of MASs that meet military requirements. This research focuses on enabling MAS design and synthesis tools like agentTool to store, retrieve, and filter persistent, reusable, and reliable agent domain knowledge. This "enabling" is vital if such tools are expected to produce consistent, maintainable, and verifiable agent applications on short timetables. Enabling requires: 1) modeling the agent knowledge domain, 2) designing and employing a persistent knowledge base, and 3) bridging that domain model to the knowledge base with an extensible domain interchange grammar. The achieved interchange grammar, called Multi-Agent Markup Language (MAML), is presented and shown to be capable of representing MAS design knowledge in a concise and easily parsed form that is readily stored and retrieved in the knowledge base. The selected knowledge base, called the Agent Random-Access Meta-Structure (ARAMS), is shown to support MAML and operate in a distributed environment that permits sharing of agent development knowledge between various tools and tool instances. Tests of MAML and ARAMS with agentTool are summarized, and related future work suggested.

KNOWLEDGE BASE SUPPORT FOR DESIGN AND SYNTHESIS OF MULTI-AGENT SYSTEMS

I. Introduction

Today's world is interconnected and the medium of that connectivity is information. Technology has allowed information of all classes to be both created and made available at an ever-increasing pace. There is significant need for certain types of information interconnection to extend throughout military infrastructure. However, with military manpower forces unable to grow at a technology-matching pace, technology itself must be creatively applied to reduce the information processing workload to a manageable level. If this is not done, potentially vital knowledge will simply be either lost or just not be made available in a timely or coherent fashion. Intelligent agents provide one solution to this problem. Agents work in the information domain and reside in the high-tech hardware whose complexity grows faster than our traditional forces. They are a natural choice to maintain our military infrastructure's superiority.

A limiting factor in this choice is that intelligent agents must interact with humans and each other in order to handle the diverse taskings that they, the agents, may be commissioned for. Additionally, security must be integrated at a ground level. These concerns merely transfer the manpower shortage to technical specialty personnel such as software programmers. Fortunately, software development aids can alleviate the manpower burden at this level.

1.1 Background

The science of Artificial Intelligence has greatly expanded since the advent of intelligent software agents in the mid-eighties. As a direct result, a burgeoning population of *intelligent agents* is finding its way into many aspects of our society. The military, and particularly the Air Force, is not, nor should be, exempt from this incursion. Both *Joint Vision 2010* and *Air Force 2025* point out accelerated movement towards distributed C3I applications and the need for security in these applications. Agents lend themselves naturally to both these elements.

Unfortunately, the complex and evolving nature of distributed software (especially multiple interacting agents) requires significant expertise. This expertise can be provided by traditional human resources or, less expensively, by using flexible software design tools. A recent research program at the Air Force Institute of Technology is developing such a tool for the rapid design of agents and agent systems. That tool is called agentTool.

1.2 AgentTool

A handful of commercial toolkits have recently become available for designing agents and agent systems (Agentsoft 1998; IBM 1999; Ndumu 1999). Many of these tools are geared for use in economic applications such as electronic commerce and inventory management. In contrast, none are well suited for the breadth of military application, especially where security is concerned. Additionally, few allow for the verification of a design's reliability that is only now becoming available for traditional non-agent software (e.g. automated debugging and formalized specification verification).

One candidate for filling this niche is agentTool, an AFOSR-sponsored, AFIT-led effort to create a Java-based tool for rapid development of multi-agent systems. Because agentTool multi-agent systems are generated in Java (currently), they will be platform-independent and

operable in a distributed environment. AgentTool considers communication, coordination, and security as critical factors up front. Other factors, such as agent mobility, may be incorporated through extensions. Systems produced by agentTool are not limited to simple information-gathering and commerce applications. Applications such as intelligent control of unmanned aerial vehicles (UAVs) and acquisition system modeling are also possible. Moreover, agentTool's design allows for formal verification of agent system specifications.

For the initial 1999-2000 research cycle, four areas vital to agentTool operation were addressed. These are:

1) Development of a Multi-Agent System (MAS) design methodology (Wood 2000)

2) Specification of an agent architecture description language (Robinson 2000)

3) Creation of a storage/retrieval structure and mechanisms (e.g. knowledge base system) to contain and manipulate reusable agent domain knowledge (this work).

4) Formal verification of agent systems (Lacey 2000)

This effort focuses on the third of these areas.

1.3 Problem

The agentTool system will help to develop secure systems rapidly and reliably. Persistence, reusability, and reliability of design are keys to facilitating this. Persistence allows agent systems and system components to be stored in long-term memory. Some elements that might require persistence include hardware interfaces, application interfaces, communication protocols, coordination schemes, security protocols, agent architectures, data structures, and system frameworks. Along with these items, more abstract concepts such as agent roles, rules, conversations, and intentions may also need to be stored. Reusability refers to making these

persistent elements available at varying levels of abstraction. For instance, agent roles are templates that may be assigned to a diverse class of agents whereas communication protocols may be very specific to a single communications framework. Both roles and protocols, however, are reusable to an appropriate extent. A structure for storing and maintaining such a wide scope of knowledge is currently non-existent for agentTool and is not readily available for most other agent construction utilities. Without some instrument to meet reusability, persistence, and reliability requirements, agentTool will be incapable of meeting its design goals. The following statement summarizes the aim of this effort in regard to agentTool requirements as just introduced:

> **Goal Statement:** *Develop a technique/methodology for determining knowledge base requirements, constraints, and contents for a multi-agent system development environment. Test the technique by producing a prototype agent knowledge base and evaluating it.*

1.4 Requirements and Assumptions

Even though the goal statement captures the general purpose of this effort, it can be partitioned and expanded to reflect more specific requirements. These requirements will be significant throughout the effort in gauging progress. Three key requirements are:

1. Research and model the agent development knowledge domain.
2. Develop a process (methodology) for structuring a knowledge base containing the domain knowledge model.
3. Produce a prototype and test it with agentTool.

Two significant assumptions apply to these requirements. The first is that the agent domain model will contain a wide range of knowledge classes, of which only a subset may be implemented and tested fully in the prototype within the time allotted for completion of this

effort. The second, related assumption is that the prototype knowledge base will be critiqued for interoperability with agentTool, persistence and reusability of content, extensibility, and reliability. Section 1.2 notes that interoperability with agentTool starts with designing the knowledgebase in Java. However, Java alone is not sufficient for interoperability. True interoperability is achieved by requiring a set of standard interfaces between the tool's components and packages so that changes in any component do not require significant changes in the other components. These Java interfaces are established through the AFIT Agent Research Group (ARG).

1.5 Overview

The remainder of this effort in composed of five chapters. Chapter 2 reviews research efforts and general knowledge pertaining to knowledge representation, knowledge base structure, agent design, and agent system design. Chapter 3 outlines the methodology for obtaining the goal stated above in Section 1.3. Chapter 4 presents the design decisions and related details of implementing a knowledge base prototype that supports agentTool and even other agent utilities. Chapter 5 demonstrates the behavior of the prototype in regards to agentTool. Finally, Chapter 6 analyzes the effort's end results and provides suggestions for future work.

II. Background

The crux of this effort involves development of knowledge support system for the agentTool multi-agent system development environment. Such a knowledge base must capture all the knowledge needed to develop diverse agent systems. In designing a knowledge base to provide this knowledge, consideration should be given to both the knowledge representation format as well as the implementation structure of the knowledge base as a whole. This chapter focuses on what research has already been accomplished in each of these three areas.

Before reviewing research in these fields, it is appropriate to define the term *knowledge base*. Though multiple definitions exist, a knowledge base is most simply defined in regards to its composition or content. From this compositional perspective, a knowledge base is a "large collection of facts, rules, and heuristics that capture knowledge about a specific domain of applications" (Schmidt 1989). Since several of the terms used in this definition reappear throughout this work, it is useful to provide some additional definitions. *Facts* and *rules* are representations of specific and general knowledge, respectively. *Heuristics* are a class of rules (rules-of-thumb) regarding relationships within a specific domain, and are usually an order of magnitude more complex than rules (Gonzalez 1993). Facts, rules and heuristics allow *inference* and *reasoning*. Inference refers to the passing from one proposition considered as true to another whose truth is believed to follow that of the former and reasoning refers to the creation of inferences from known facts to bring about coherent and logical thought (Webster 1986).

Though accurate, the above definition of a knowledge base contains no notion of how knowledge bases fit into the rest of computer science and why they are needed. This is resolved by expanding the former definition as follows:

> A *knowledge base* is a product of knowledge engineering, which is a discipline within the science of artificial intelligence. Artificial

> intelligence, in turn, is a branch of computer science. Computer science, together with its peers: philosophy, psychology, and linguistics, form the cognitive sciences. Each of the cognitive sciences studies the mind or understanding in a different way. Computer science stands out in this group because it involves mimicking the mind or its processes via computer programs rather than simply analyzing them. Because traditional computer science had no effective mechanism to imitate important human characteristics (e.g. learning, reasoning, and self-correction), the sub-science of artificial intelligence was created. AI programs use a changing knowledge base rather than fixed, pre-programmed algorithms to simulate human behavior (Lukose 1996).

This expanded definition places the idea of domain-specific knowledge into the context of software that focuses on imitating humans. However, it does not make sense to have such a *virtual person* without a means to manage and apply its knowledge. Therefore it can be seen that knowledge bases are not intended to be stand-alone software entities, but rather an integral part of some domain-specific knowledge-based system. In particular, both expert systems and AI knowledge-based management systems (KBMS) utilizes some sort of reasoning engine and meta-knowledge in order to function (Gallaire 1989; Williams 1990).

The remainder of this chapter provides an overview of past and current efforts in knowledge base design and agent domain research. Specifically, Section 2.1 reviews several proposed knowledge representations schemes. Section 2.2 then introduces options for overall knowledge base structure. Section 2.3 focuses on work done in defining and populating the agent domain space. Section 2.4 summarizes the previous three sections and provides a transition to Chapter 3.

2.1 Knowledge Base Representation Schemes

As discussed above, a knowledge base may contain knowledge as facts, rules, or heuristics. Though correct, this is very general. Several perspectives exist to explain the different ways that these facts, rules, and heuristics may be specifically represented in a knowledge base.

This section gives a brief comparison of these perspectives and their corresponding representation schemes.

Gonzalez and Dankel categorize knowledge representation schemes into five areas: logic, rules, semantic networks, frames, and objects (Gonzalez 1993). Nixon adds a sixth category for Entity-Relationship representation (Nixon 1989). Other authors generalize these six facets into two or three broader categories. Amilcar and Cristina Sernadas visualize logical, structural, and procedural paradigms (Sernadas 1989) whereas Lukose, Kramer, and Pedersen take a black-and-white stance where knowledge is either represented declaratively or procedurally (Lukose 1996; Pedersen 1989a). Which of these three perspectives is best depends on the level of abstraction required. Since Gonzalez and Dankel provide the greatest depth and breadth of information, their classification is detailed first, with references to the other two generalizations made as needed.

2.1.1 Logic

Formal first-order predicate logic has been at the core of artificial intelligence since its infancy. Because of the rigorous mathematical base and the Boolean base of this logic, computers are able to use it naturally. Knowledge engineers often hold first-order predicate logic as the preferred language of knowledge base construction because it can be used both to represent knowledge and reason over that knowledge. With logic, the knowledge base is considered as a *theory* (set of theorems) obtained by varying a set of stored facts in a *fact base* (logic axioms) using a fixed set of logic inference mechanisms (Sernadas 1989). Reasoning takes place by applying certain inference mechanisms in an inductive, abductive, or deductive form, to generate new facts (Gonzalez 1993). For example, given $(A \vee B)$ and $(B \vee C)$, the new fact $(B \vee (A \wedge C))$ may be derived using available logic inference constructs.

There are a number of limitations to using logic in the realm defined above. One of these limitations is the lack of mechanism to remove axioms and facts that become invalid due to some

other change in the knowledge base. This is a specific instance of the more general problems of lack of organization principles and absence of inference procedure control (Sernadas 1989). Logic is also inflexible due to its implicitly declarative language, which does not allow much flexible application of reasoning methods (Gonzalez 1993). Fortunately, there are alternatives to using pure logic representation in a knowledge base.

2.1.2 Rules

Populating a knowledge base following a rule paradigm is the classical approach used in expert systems. A medical diagnosis expert system, for example, might maintain a set of if-then rules that relate symptoms to diagnosis. An inference engine would attempt to pattern-match a symptom with the left-hand side of rules in the rule set. A match would cause the return of the right hand side of that rule, which will either be the diagnosis or a match for the left-hand side of yet another rule. In this second case, multiple rules may then "chain" to a diagnosis.

Though rule-based inferencing is founded in logic, there are several distinctions that make rules more flexible than logic alone. An example of this *forward chaining* is shown and explained in Figure 1. Alternatively, rule-based systems may use *backward chaining*, which is also called *goal-driven* reasoning. There are two rule-based architectures for supporting these reasoning methods: *inference networking*, and *pattern matching*. The first considers the knowledge base as a graph where rules are represented by edges and nodes represent facts. The second, pattern-matching, tries to match facts to the patterns of either the RHS or LHS of rules, depending on whether forward or backward chaining is being used. A great deal has been written on additional features of these two architectures (Gonzalez 1993).

Compared to logic-based representation, rule-based representation offers greater ease of use (especially in populating the knowledge base) and an escape from the restrictions of strictly Boolean results. However, the rules approach has three drawbacks: *infinite chaining*,

contradictory knowledge, and *behavioral opacity*. Infinite chaining occurs when a set of inferences loops back on itself in a way so that the rules are repeatedly checked with no way to terminate. This problem exists between these rules:

$$\mathbf{A} \Rightarrow \mathbf{B}$$

$$\mathbf{B} \Rightarrow \mathbf{C}$$

$$\mathbf{C} \Rightarrow \mathbf{A}$$

Contradictory knowledge occurs, as its name suggests, when the addition of a new rule has an undesirable side effect of invalidating a previously correct inference path. Finally, the nature of rule-based representation makes it behaviorally opaque, which means that determining when rules will fire is not easy. This causes problems in debugging and maintaining large rule-based knowledge bases because, even when all rules are formed correctly, the sequence of their execution may lead to hard-to-detect errors (Gonzalez 1993; Pedersen 1989a).

	---Left-Hand Side (LHS)---	⟶	**---Right-Hand Side (RHS) ---**
Rule 1:	`If a program is mobile`	`THEN`	`the program is an agent`
Rule 2:	`If an agent has goals`	`THEN`	`the agent is goal-based`

Given the following facts: `Program X is mobile and has goals`

A rule-based system deduces: `Program X is a goal-based agent`

Figure 1: Rule-Based Example for Agent Application Domain

A partial solution to rule and logic limitations involves applying structure to the knowledge base. Doing so generates a structural knowledge representation (as opposed to the strict logical representations so far discussed). Sernada and Sernada point out that the structural approach involves organizing the fact base in a way that keys on having abstractions for related groups of facts (Sernadas 1989). Instead of using the axiomatic approach to find new facts that rules and logic take, a set of rules is applied to these *semantic abstractions* (also called *semantic*

primitives) to generate new facts. Semantic nets and frames are the two most often used structural representations.

2.1.3 Semantic or Associative Nets

Associative nets are enriched versions of the directed graphs used by the inference network architecture of rule bases. The 'richness' is arrived at by having the nodes of the graph represent concepts and objects rather than just facts, and by having edges define relations or associations between these objects rather than just rules. By doing this, the unrelated facts of rule-based representation are supplanted by explicitly and implicitly interrelated concepts. For example, associations `PART-OF` and `IS-A` might be dscribed as follows for a knowledge base.

```
(sub-structure PART-OF super-structure)
(transitive PART-OF)
(transitive IS-A)
```

By utilizing such associations in an agent domain space, queries such as *"Is a planner part of a multi-agent system?"* can now be answered (see Figure 2). This example is fairly simple, but demonstrates the power of adding semantic structure to a knowledge base. Instead of simple IF-THEN constructs, queries such as exemplified above, can be posed. One can imagine that adding more diverse relations that combine multiple concepts such as (`age IN-RANGE-OF integer1 integer2`) can make associative networks much more powerful than simple logic or rule-based representations. Even so, associative networks have drawbacks.

The drawbacks of this approach involve *frame-of-reference confusion* and *combinatorial explosion* (Gonzalez 1993). Frame-of-reference confusion arises when there is greater detail in an object being related than in the relation itself. This can be better understood by examining the shaded parts of Figure 2. If the knowledge base is queried whether `ALPHA` is part of `WORKGROUP` the response will be true because `MOBILE-AGENTS` are agents, agents are parts of a `MULTI-`

AGENT_SYSTEMs, and WORKGROUP is a MULTI-AGENT_SYSTEM (MAS). This seems fine unless ALPHA is actually PART-OF some other MAS (not WORKGROUP) or is stand-alone. Exactly what the associative net is missing in this and similar cases is a frame-of-reference, or constraint that allows for an association not to exist in particular instances.

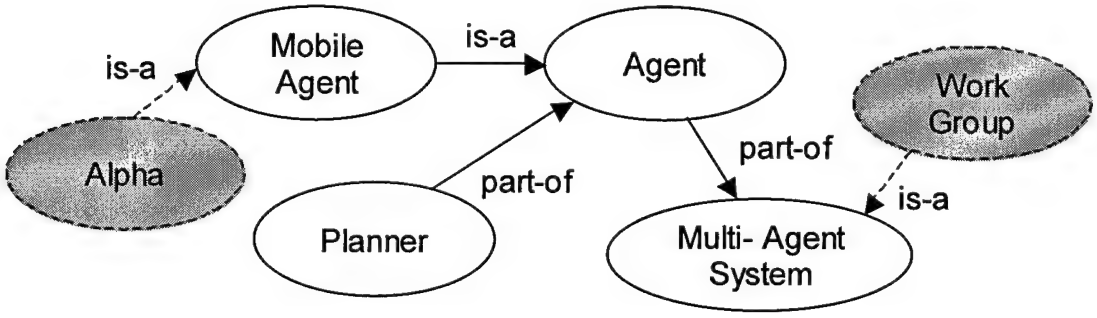


Figure 2: Associative Net Example for Agent Application Domain

The second drawback to associative networks, combinatorial explosion, occurs when an ambiguous relationship is determined false. In the AGENT example, for instance, the query: *"Is a database part of a multi-agent system?"* would receive a false reply, but only after the system had fired every rule. This is a more significant problem in large knowledge bases because of the greater number of relations to examine.

2.1.4 Frames

In 1975 Minsky tackled the associative networks' frame-of-reference problem by proposing the frame representation scheme (Minsky 1975). With frames, an entity is classified in terms of a set of attribute-value pairs that populate *slots* in that entity's *frame*. The values of these pairs, in turn, may be composed of *facets* that are constraints or operators on the attribute of that pair. For instance, in the AGENT frame shown in Figure 3 there are slots filled with attributes for Planner, Knowledge-Store, and Security-Level. For Security-Level, there are two facets: Range and If-Changed. Range constrains the value associated with Security-

Level while If-Changed references a procedure to execute if the value changes. Procedures, such as If-Changed, are referred to as *demons*. Demons are unique in that they represent procedural knowledge and are thus imperative, not declarative. The power of frames lies in both this use of demons and *frame inheritance*. Frame inheritance is captured by the Specialization-of and Generalization-of fields of a frame. In Figure 3 AGENT is a specialized PROGRAM, which is another frame not shown but which likely has slots for Programmer and Language. Because AGENT is a specialization of PROGRAM, it would inherit these slot fields implicitly. Extending this further, MOBILE-AGENT and INTERFACE-AGENT also inherit those program slots as well as AGENT's slots because they are specializations of AGENT.

```
Generic AGENT Frame
Specialization-of : PROGRAM
Generalization-of: (MOBILE-AGENT, INTERFACE-AGENT)

Planner:
      Range: (FORWARD-CHAINING, BACKWARD-CHAINING)
Knowledge-Store:
      Range: (Database, Flat-File)
Default: Database

Security-Level:
      Range: (SECRET, UNCLASSIFIED, TOP-SECRET)
      If-Changed:(ERROR: No permission to change classification)
```

Figure 3: Frame Example for Agent Application Domain

As with other schemes, frames do have disadvantages, although significantly less serious ones. The primary fault with frames is they do not represent heuristic knowledge as easily as rules. Frames are also limited by an inability to accommodate new situations or objects (Gonzalez 1993). Additionally, structural knowledge representation approaches such as semantic nets and frames suffer from a need for inference engines that are dependent on the particular scheme's domain semantics (Sernadas 1989). This is the opposite problem imposed by rule and logic-based approaches, where the inference process is encapsulated in a relatively inaccessible inference engine. One way around this close semantic binding is to merge logical and structural

representations in order to maintain a generic logic-based inference engine while keeping a more flexible representation scheme. Schemes that take this approach are known as hybrids and will be discussed in Section 2.1.6.

2.1.5 Objects

Object representation takes the procedural usefulness of frame demons a step further. Instead of referencing independently maintained procedures for a given attribute, objects *encapsulate* procedures together with the data they operate on. In the large, the object-oriented approach to knowledge representation offers the same benefits of abstraction, encapsulation, polymorphism, and inheritance that it does for programming in general. Excluding inheritance, each of these benefits is unique to the object approach and makes it more efficient than frames. However, Gonzalez points out, object-based knowledge representation has at least one shortcoming in common with frames: the accommodation of new situations (Gonzalez 1993). Forms of the Entity-Relationship scheme, another common representation form, have significant commonalities with the object approach.

2.1.6 Hybrids and Other Schemes

Whether logic, rules, associative nets, frames, or objects is best for knowledge representation depends on the application. Maybe none of these schemes (in their pure form) are appropriate in certain circumstances. In these cases, hybrid schemes may be necessary. These hybrids may be a simple merger of two or more pure schemes (Pedersen 1989b), or they may be complex enough so that they cannot be simply classified as a simple merger. Examples from both categories of hybrids are presented below.

2.1.6.1 Rule-Frame Hybrid

Implementing a rules paradigm using frames is a hybrid representation scheme for an inference net developed by Fox in 1983 (Gonzalez 1993). The purpose of using this hybrid would be to perform the functions of a rule-based system but with structural advantages. One problem a rule-frame approach alleviates is that rule firing is now controllable via the frame demons rather than being accomplished in a black-box inference engine. Additionally, this hybrid give frames the capability of suggesting other frames when found in situations that don't apply to themselves (Gonzalez 1993).

2.1.6.2 Extended Institutional Hybrid

In 1989 Sernadas and Sernadas published a framework that supports logical, structural, and procedural representation paradigms. Their extended *institutional* framework centers on considering knowledge bases as theories and individual representation schemes as parameterized theories called *theory mappings.* In such, theory mappings are semantic primitives that map one theory to another. The advantage this brings is two-fold: 1) new theories can be made from logic applied to existing theories and 2) since new theory mappings can be made, there is no restriction of using only semantic constructs of one representation scheme. In short, this framework introduces a meta-knowledge representation scheme that allows for flexibility of representation as well as of content represented. A lengthy example of creating a theory mapping for an Entity-Relationship representation approach is given by the Sernadases (Sernadas 1989). Unfortunately, the extended institutional framework, despite its extensive advantages, lacks real-world validation (especially of the vital theory mappings).

2.1.6.3 CKML, OML, and XML

The Conceptual Knowledge Markup Language (CKML) is one of the many proposed application-specific extensions to XML (eXtensible Markup Language), the data format standard

for structured document interchange on the Web (Olsen 1997). Since CKML, like its more-narrowly-scoped cousin the Ontology-Markup Language (OML), extends XML to allow for broader content-based access to documents and related entities, it may be utilized in the representation of knowledge entities in a knowledge base. The two most powerful aspects of CKML are 1) it is text based, and 2) it allows knowledge to be accessed via multiple *facets*. The concept of a facet is roughly equivalent to that of an ontology; meaning, it identifies a frame of reference for an object. The text based grammatical constructs of CKML enable multi-faceted access to knowledge by forming class hierarchies called *categories* (using IS-A). Knowledge content then takes the form of objects that are instances of one or more ontology categories. Each category is accessible though a facet (Olsen 1997). Unfortunately, the majority of available literature on CKML relates to the language and falls short in detailing extensive knowledge base application.

2.1.6.4 Abstract Syntax Trees

CKML's XML foundation allows it to be represented by a *formal grammar*, which provides consistent syntax for developing knowledge-based applications and for representing knowledge. This idea of having a formal grammar has merits in the application of yet another knowledge representation scheme - the abstract syntax tree (AST). The following are two complementary definitions for AST:

> **AST-** A data structure representing a program which has been parsed, often used as a compiler or interpreter's internal representation of a program while it is being optimized and from which code generation is performed. The range of all possible such structures is described by the abstract syntax (English 1998).

> **AST-** This is a tree structure, typically an N-ary tree, that mirrors the abstract syntax of the source language. Each leaf in the tree corresponds to a terminal of some sort in the language (say, a constant or a variable) and each node corresponds to an operator or, perhaps, a non-terminal (add, subtract, assign). This type of structure is typically the result of a parse, and is used for type

checking and as input to a later stage of the compilation process (VandenBerghe 1999).

The power of ASTs is in their abstract nature. A single AST can be reused in multiple ways just by redefining how it is interpreted. ASTs are created by *parsing* special domain-defining programs written in a formal grammar (as CKML has) into a tree of nodes according to tree-building algorithms. Once created, these *Domain ASTs* are used to guide specification of specific programs in the AST's domain. Succinctly put, a program's specification knowledge is parsed into an AST to be either read, modified, or translated by special "visitor" or *tree-walking* programs. Compilers, for example, perform this function by applying certain tree-walking algorithms to parsed program code in order to optimize or compile the code.

Although visitors provide important functionality, it is the ASTs themselves that declaratively represent knowledge. Visitors merely provide a mechanism for interpreting the AST representation. The same applies to all knowledge representation schemes; a physical mechanism for maintaining or operating on the represented knowledge is required for a scheme to be useful. The following section discusses information relevant to designing such a mechanism.

2.2 Knowledge Base Implementation Structures

At the beginning of this chapter a definition was given for *knowledge base*. A number of authors, including Pedersen, add the following simple addendum to that definition; that is, a knowledge base is, at a fundamental level, a database of knowledge (Pedersen 1989a). Understanding this is the first key to seeing that a database structure is the first logical option for knowledge base implementation. However, there are various classes of database. Understanding which is best for a given need requires an understanding of the fundamental differences between them. The remainder of this section will describe several common data base structures, particularly their strengths and weaknesses.

Databases are the key elements to database systems (DBS). Database systems collect in one package not only the basic storage structure, but also access and control mechanisms for that structure. Modern DBSs usually have three levels of architecture. The first level, and closest to physical storage, is called the *internal* level. Above it is the *external* level, which is the interface to the user or application. Between these two exists the third, or *conceptual,* level. Each of these levels presents a certain *view* of the database, which is established by what are called *schemas*. The three schema types (one per level) can be defined as follows.

> ***Internal Schema***: A representation-dependent description of the database corresponding to a precise specification of the storage structures and access methods used to store data in secondary memory (Galdarin 1989).
>
> ***Conceptual Schema***: A collection of descriptions of stored data written in a storage-structure-independent way. External schemas are written in terms of the conceptual schema.
>
> ***External Schema***: A description of a part of the database corresponding to a program or a user view of the modeled [domain]. The nature of the external schema prevents a user from modifying database content not specifically permitted by the schema, e.g. data security (Galdarin 1989).

These various schemas are written using Data Description and Data Manipulation Languages (DDLs and DMLs).

The database system architecture just described follows what is called *conceptual data model design*. Traditional *hierarchical* and *network* database systems do not fit this architecture because they do not permit the inclusion of conceptual schemas. *Relational* and *object-oriented* (OO) data models, however, do. By allowing for conceptual schemas at least three advantages arise. First, community sharing of a database storage structure becomes feasible. Second, formal representations of user-scoped data and relationship views are permitted. Finally, the data domain becomes enriched (more detailed). This enrichment takes the form of (1) a set of real-world entities that correspond to stored data elements, (2) a set of relationships between these entities, (3) a set of attributes belonging to these entities and relationships, and (4) specific

properties of entities, relationships, and attributes, such as the cardinalities of relationship participation (Galdarin 1989).

Of the four classes of database system introduced above, network and hierarchical systems are simplest but lack the additional schema that relational and OO systems use. As a result, relationships that have attributes are not easily captured. Additionally, many-to-many relationships cannot be readily modeled. Although special data records may be added to get around this to some extent (in the network model), doing so slows data access speed down while adding complexity. Predicate logic is the data description language of both network and hierarchical data models (Brodie 1989). The primary difference between the network and hierarchical data models is that the former organizes data as a directed graph net while the latter models it as a tree structure. By imposing a strict tree structure, any given data element in a hierarchal model can only have one *parent* element. This leads to significant data redundancy (multiple hierarchies) when more complex relations need to be modeled. In cases when none of the above-mentioned complex relations are needed for a given application, either the hierarchal or network database system models provide the fastest and most efficient DBS model solutions.

As already mentioned, the conceptual schema permits modeling options in relational and OO models that are not as available in the non-conceptual models. Beyond this, the relational model captures both entities and relationships as data objects (the basic non-conceptual models use pointers for relationships). It does this in tables in a way that permits mathematical precision in the operation of the database system (using *relational algebra* and *relational calculus*). In fact, its formal mathematical underpinnings are a key strength to the relational model, permitting such activities as direct capture of a design into the database tables (Bjork 1998b). The contents of the relational table generally appear as a set of ordered atomic tuples, such as:

UAV_TASKING=(UAV_ID:integer, TASK:string)*. These tuples can be reused to limit the need for the data redundancy that is present in the hierarchal model. Relational systems do have some weaknesses however. First, they are somewhat slower than non-conceptually modeled systems. Additionally, they have difficulty supporting non-textually represented data. This also includes representing behavioral information that accompanies certain data types. Object-oriented systems overcome both these weaknesses by storing more complex forms than atomic tuples. Saving complex structures saves time by avoiding the relational step of decomposing complex entities for storage as atomic tuples. Behavioral information is also stored in encapsulated form with the data it applies to. Another thing that the OO model handles better than other models is polymorphism, the allowance of variations in the inheritance relationship. The key ingredient present in the relational model that does not carry over into the OO model is the inherent mathematical basis. Because of this, proposals to extend the relational model to include non-atomic data types and behavioral knowledge have been made (Bjork 1998a).

Hierarchal, network, relational, and OO DBSs have been implemented in virtually every sector, public and private. Some implementations follow one of these models exclusively while others provide partial implementations, using only what is needed. Tools exist to facilitate going in either of these directions. Oracle databases, for instance are generally relational and the DDL and DML they use (SQL) makes effective use of relational algebra and calculus. At the opposite end of the spectrum there are languages such as CKML (Section 2.1.6.3), which are not generally associated with any one vendor or underlying structure at all, but which are still useful as DDLs or DMLs. In the agent domain specifically are several tools that use databases for storage. Bit & Pixels IAFactory takes a traditional approach by storing an agent in terms of a specification containing the agent's state table information. This is a simple approach but seems limited in the

* This defines that an relationship called UAV_TASKING is composed of a UAV_ID and a TASK.

domain of agents that can be created this way (Thomas 1999). AgentSoft's LiveAgentPro takes a step away from this by using JavaScript scripts (Agentsoft 1998). Taking these approaches greatly simplifies knowledge base (database) creation but only at the cost of having less flexibility than more intricately modeled databases.

2.3 Knowledge Base Content

Since any given knowledge-based application operates in a specialized knowledge domain, that domain should be well understood before choosing any representation scheme whether pure or hybrid. This section explores agent and multi-agent system composition in order to lay a foundation for later selection of an appropriate knowledge representation scheme. To accomplish this exploration, a variety of efforts aimed at analyzing the agent domain will be reviewed.

2.3.1 Franklin and Graesser

It is important that a definition for agent be given here in order to scope the extent of this investigation. Franklin and Graesser provide a thorough comparison of several agent definitions and conclude this comparison with the following definition:

> An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda so as to effect what it senses in the future (Franklin 1996).

This definition is broad enough to even include non-software agents but specific enough to distinguish agents from programs. From the definition it is already apparent that agents are characterized by at least two elements: properties and components. Properties include *goal-driven* (pursues agenda), while components include Application Programming Interfaces (API) and *state machines* (acts on environment and senses time change). In addition to the low-level properties, there appear to be one or more high-level properties (e.g. *autonomous*) for classifying

whole categories of agents. A collection of high-level and low-level agent properties collected by Franklin and Graesser appear in Table 1. According to the given definition, all autonomous agents will have the first four properties listed in the table. However, any single agent may also have one or more of the remaining properties.

Table 1: Agent Properties (Franklin 1996)

Property	Other Names	Meaning
Reactive	Sensing and acting	responds in a timely fashion to changes in the environment
Autonomous		exercises control over its own actions
Goal-oriented	Pro-active/purposeful	does not simply act in response to the environment
Temporally continuous	Persistent	is a continuously running process
Communicative	Socially able	communicates with other agents, perhaps including people. Requires protocols/transports.
Learning	Adaptive	changes its behavior based on its previous experience
Mobile		able to transport itself from one machine to another
Flexible		actions are not scripted
Character		believable "personality" and emotional state

Franklin and Graesser also discuss classifying agents into a taxonomic structure. Figure 4 shows the highest levels of Franklin and Graesser's agent *taxonomy*. The subclass covering software agents might be further classified by control structure, environment, programming language, etc. For example, classification by control mechanism could produce subclasses using planning, regulation, and adaptive mechanisms. Binary classification is another taxonomic classification option. Under this approach, the properties listed earlier may be the basis of classification. For instance, agents may be categorized as mobile or non-mobile, learning or non-learning, etc. This would quickly generate a large (and redundant) binary tree from a pool of features or properties like those already discussed. Franklin and Graesser also discuss classification methods adopted from the sciences of mathematics and psychology. One of these schemes, matrix organization, considers each of n features/properties as a dimension in n-

dimensional matrix space, giving each matrix cell a unique category of classification (and removing redundancy). This allows for absolute identification of mobile agents, mobile communicative agents, mobile communicative learning agents, and so on (Franklin 1996). The key assumption of this scheme is that the properties are well defined.

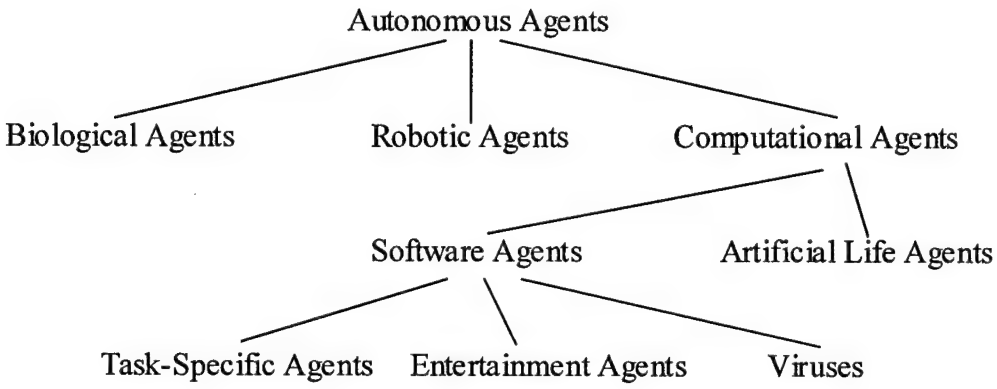


Figure 4: Upper Levels of an Agent Taxonomy(Franklin 1996)

2.3.2 University of Michigan

Working independently but in parallel with Franklin and Graesser, a University of Michigan (UM) effort also produced a property listing for agents. The UM list, however, encompassed significantly more detail by adding further property descriptions and a limited property hierarchy (see Table 2). Beyond confirming Franklin and Graesser's consideration of properties, the UM effort gives more substantive support to these same researchers' indirectly mentioned idea of agent component composition. In particular, properties are noted to be of little use without some corresponding implementation in the form of architectures and/or components *(Arriola 1994)*.

The UM study that produced the information in Table 2 did so by analysis of several specific agent systems. In that analysis, it was noticed that many systems had properties in common despite having very different supporting implementations of them. The differences were accounted for by looking at each system as a collection of agent components forming an agent

architecture. Table 3 lists the architectures categorized under this premise, as well as their corresponding properties.

Table 2: Subcategories of Three Key Agent Properties (Arriola 1994)

Property	Sub-property	Meaning
Reactive	Prediction	Determining what changes in world may occur as result of
	Query-answering	Decision describing. Agent can explain how it arrived at a conclusion.
	Language Perception	Can send and receive words to communicate.
Goal-oriented	Planning	Based on establishing a set of actions to achieve a goal.
	Re-planning	Modifying or rebuilding plans because of environmental changes
	Simultaneous Multiple-Goal support	Can work toward several goals at once
	Self Reflection	Can examine own behavior by actively applying meta-knowledge on internal mechanisms
	Meta-Reasoning	Relates to skill improvement, adaptation, and learning. Can be deployed implicitly through mechanisms such as domain-independent learning, or explicitly using, for example, declarative knowledge, which the agent can interpret and manipulate.
	Deductive Reasoning	Takes form of *IF A THEN B*. Basis of explanation-based learning. Gives no new semantic information.
	Inductive Reasoning	Allows for developing new semantic knowledge.
	Expert System	A rule-based rather than planner-based agent
Learning	Single Method	Self-describing
	Multi-method	Agent can use more than one learning method
	by Instruction	Agent is given domain knowledge when it asks or because it is "educated" by a teacher.
	by Experimentation	Discovery. Applying perceptions to domain knowledge to refine it.
	by Analogy	Reasoning by analogy generally involves abstracting details from a particular set of problems and resolving structural similarities between previously distinct problems. Analogical reasoning refers to this process of recognition and then applying the solution from the known problem to the new problem. Such a technique is often identified as *case-based reasoning*
	Transfer of Learning	Learning can be passed between agents
	Inductive learning and Concept Acquisition	*Concept acquisition* refers to the ability of an agent to identify the discriminating properties of objects in the world, to generate labels for the objects and to use the labels in the condition list of operators, thereby associating operations with the concept.

	By Abstraction	Contrasted with concept acquisition, *abstraction* is the ability to detect the relevant -- or *critical* -- information and action for a particular problem. Abstraction is often used in planning and problem solving in order to form a condition list for operators that lead from one complex state to another based on the criticality of the precondition.

Table 3: Some Agent Architectures (Arriola 1994)

Architecture	Features
Subsumption Architecture	No planning, "instinctive"
ATLANTIS	Planning and Re-planning; Multiple-Goal;
Theo	Learning by Concept Acquisition; Planning; Multiple-Goal; Self-Reflection; Meta-Reasoning; Prediction; Language Perception;
Homer	Planning and Re-planning; Multiple-Goal; Self-Reflection; Language Perception; Language Perception;
Prodigy	Learning by Instruction, Analogy, Abstraction, and Experimentation; Planning; Self-Reflection; Meta-Reasoning; Prediction;
Soar	Learning by Instruction, Abstraction, Analogy, and Concept Acquisition; Transfer of Learning; Planning and Re-planning; Meta-Reasoning; Expert-Capable; Language Perception;
Teton	Planning; Multiple-Goal;
RAPLH-MEA	Planning and Re-planning; Multiple-Goal; Meta-Reasoning; Prediction;
Entropy Reduction Engine	Learning by Concept Acquisition; Planning; Prediction;
Meta-Reasoning Architecture	Learning by Experimentation; Planning and Re-planning; Self-Reflection; Meta-Reasoning;
Adaptive Intelligent Systems	Planning and Re-planning; Multiple-Goal; Self-Reflection; Meta-Reasoning; Expert-Capable; Prediction; Language Perception; Reasoning processes for diagnosis, prediction, and planning; real-time control; global coordination of multiple tasks; reasoning by analogy; learning from experience.
ICARUS	Learning by Concept Acquisition; Planning and Re-planning; Self-Reflection; Prediction;

2.3.3 McGill University

In contrast to the UM work, which identified properties and associated architectures in existing agent systems, some groups sought to develop a universally applicable architecture. One example is the Unified Agent Architecture (UAA) project at McGill University(Belgrave 1995). McGill's generic UAA is intended for creating any agents, which contrasts the UM effort's categorization of existing ones. Even so, UAA work required some agent domain analysis in

order to extract common agent mechanisms that would need to be captured by UAA's generic agent class. The results of this analysis produced a UAA agent with the following composition: 1) an *Execution Facility* that often contains an finite state machine (FSM) and model of the world that the agent operates in, 2) a *Communications Facility* which holds the standard protocol set for information exchange with users, resources, and other agents, 3) a *Transport Facility*, which is a process set allowing agent to change domains, both local and remote, and 4) a *Packaging Facility*, which is needed for mobility and persistence (Belgrave 1995). These four facilities are components associated with the UAA architecture. In devising UAA, its authors realized the need for a broad agent *framework* that allowed for agents to be constructed on a variety of platforms. CORBA was chosen as the framework for UAA.

2.3.4 University of Cincinnati and Stanford

While the University of Michigan sought to categorize existing agents and McGill University attempted to abstract common features from them, others focused on scoped analysis of inter-agent activities. Two of most prominent works in this area were accomplished at the University of Cincinnati and Stanford (Frost 1996; Guha 1994). Both involved analyzing the communication needs of agents and creating a generic framework for meeting these needs. Both Stanford's JATLite and University of Cincinnati's JAFMAS frameworks model inter-agent communications as message-passing conversations. JATLite, however applies a TCP/IP substructure while JAFMAS is based on Java RMI.

2.3.5 Elizabeth Kendall

Kendall's work parallels that of the UAA group at McGill. That is, she focuses on abstracting a common architecture for agents by first analyzing multiple existing agents and agent systems. However, the result of her work, the Royal Melbourne Institute of Technology (RMIT) agent framework, is significantly different than McGill's UAA. In fact, the RMIT agent domain

analysis does not seem to directly capture architectural components at all. Instead, the architecture is based on using a set of interacting *layers* rather than objects. These layers model behavioral domains of agents but not the structural characteristics captured by components. The seven key layers (and their accompanying behaviors) appear in Figure 5. Each layer of the model interacts only with immediately adjacent layers. Within each layer, a *design pattern* exists to impose a functional structure on that layer. In the design patterns are certain *abstractions* that relate to the components mentioned in Sections 2.2.3 - 2.2.4. For example, the sensor abstraction that is part of the design pattern for the RMIT *Sensory* layer may actually map to a component object. All component (structural) objects in RMIT are generic and are captured in a static configuration independent object (CIO). It is through the CIO that a realization structure is provided for agents modeled behaviorally in the RMIT framework. Unfortunately Kendall provides little detail on CIO internals (Kendall 1998b).

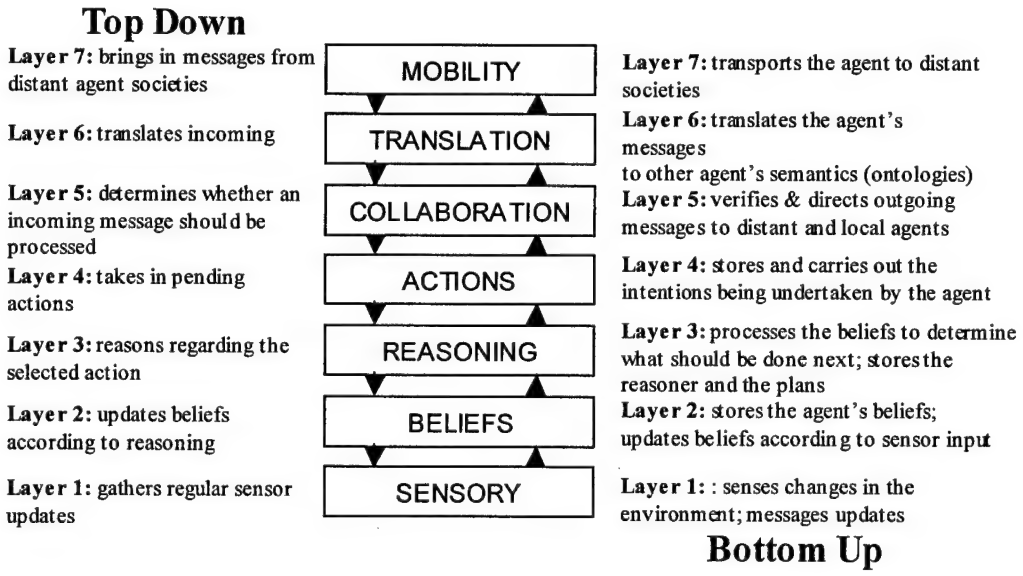


Figure 5:RMIT Agent Framework (Kendall 1998b)

The idea of separating agent behavior from agent structure is not unique to design patterns. Agent *roles* and *role models* also capture agent behavior independent of agent structure.

According to Kendall, "a role focuses on the position, responsibilities, and collaborations of an entity within an overall structure or system." (Kendall). This is independent from what would normally be captured in an agent modeled as an object since an object primarily captures capabilities. Another difference between the objects and the roles is that objects are instantiated from classes while roles are elements of subsystems instantiated from *role models*. Figure 6 shows the relation between a particular set of agent objects and an agent role model. This particular role model is composed of two roles, which are used by several different agents. The importance of this is that elements of agent systems not captured by other research efforts (i.e. behavior) are captured by role and role models.

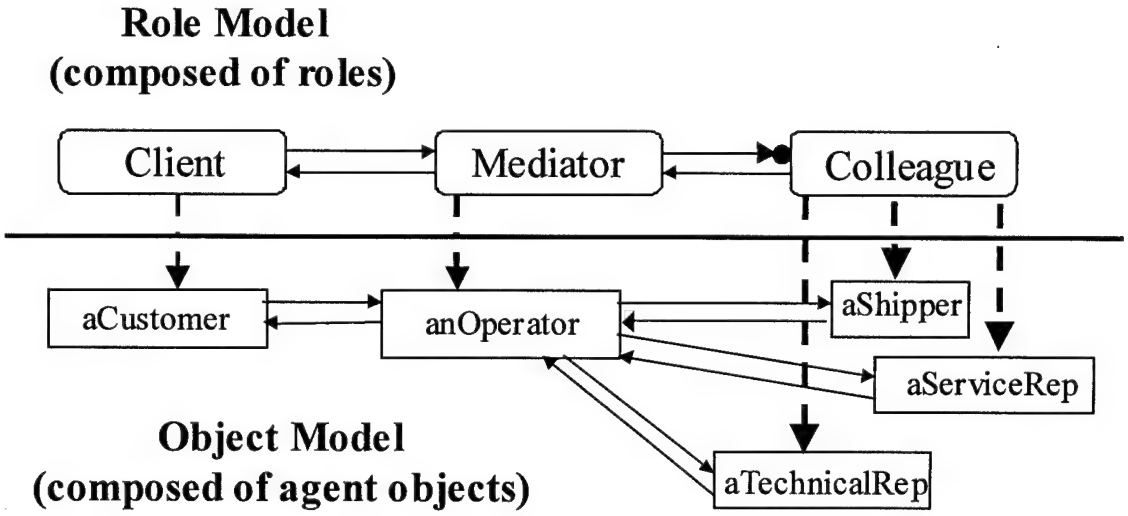


Figure 6: Relationship between Role Model and Object Model

2.3.6 FIPA

In addition to research done in classifying agent properties, components, roles, role models, and architectures, work is progressing in defining other aspects of agent space. The Foundation for Intelligent Physical Agents (FIPA), for one, is doing exactly that (FIPA 1999b). After FIPA formed in 1996, work started on standardizing agent management, agent naming, and

agent communications. Draft specifications similar to ISO standards were the products of this work. In 1999 FIPA expanded their focus to include more than negotiation, communication, and inter-operational issues. Specifically, they began formalizing how to construct multi-agent systems using software engineering principles. In this area, two significant drafts have been released. The first is a draft standard for agent attributes or "abstractions" (FIPA 1999a). The second is an architectural overview of the FIPA abstract architecture (FIPA 1999b). Both will be discussed here.

FIPA points out the purpose of the first of these two papers is to resolve the following four agent domain classification/specification problems:

1. How to have alternative mechanisms (components) for particular functions (properties and attributes). For example, the enabling of multiple transport mechanisms for messages and allowing for alternative representations and encoding of content languages.

2. How to map or integrate FIPA's specifications with existing or emerging technologies. This includes such diverse technologies as XML, SMTP, Java, Jini, Active Objects, CORBA, Directory models, Web servers, e-commerce and various messaging transports.

3. Defining conformance models and conformance tests that enable interested parties to verify conformance to FIPA specifications

4. Defining and describing levels of interoperability. The issues of interoperability include the relationships between agents, the relationships between agents and platforms, the relationships between different implementations of agent services, and versioning issues (FIPA 1999a).

These are all significant issues that have not, as of yet, been adequately addressed by the agent development community. Even FIPA's own effort in these areas has only produced preliminary results. What has been accomplished is a cursory specification of three dimensions of the agent domain: attributes, abstractions, and relations. The following lists several FIPA agent domain attributes and abstractions.

- Agent
- Name
- Service
- Address
- Agent-platform
- FIPA-message

- Message-transport-service
- Agent-host
- Message-content
- Message-content-language
- Message-encoding-representation
- Ontology
- Ontology-service
- Content language expressions
- Agent-communication-language
- Directory-service
- Naming-service
- FIPA-message

Though this is useful information, questions remain. For instance, are *services* capturing the same information as Kendall's roles?

In addition to defining agent content by attributes, FIPA's effort has resulted in classification of many agent *relationships*. These relations are predominantly ones that exist between the agent abstractions just introduced. Table 4 reflects some of the proposed relationships or associations between agent attributes and other abstractions.

Table 4: Some Agent Relationships (FIPA 1999)

Architectural Abstract Concepts and Relations
Agent associated with one or more **agent-platforms**
Agent has a **name**
Agent has an **address**
Agent can send an **FIPA-message**
Agent can send a **FIPA-message** to one or more **agents**
Agent can provide a **service**
Agent-platform can host one or more **agents**
Agent-platform provides one **naming-service**
Agent-platform provides one or more **directory-services**
Agent-platform provides a FIPA **message-transport-service**
A **directory-service** provides a mapping between **agents** and one or more **services**
The **naming-service** provides a mapping between **agent-names** and its **address**
A **message-transport-service** supports one or more **FIPA-transport-protocols**
An **FIPA-message** has a **content**
An **FIPA-message** is sent by an **agent**
An **FIPA-message** is received from an **agent**
Content is expressed in a **content-language**
Content may reference one or more **ontologies**
Content is either **action, proposition** or **object description**
Content-language must be capable of expressing at least **actions, propositions** and **objects**
Service may be provided by an **agent**
Service has a **service-name**
Service can declare its **service-interface**
A **service-interface** can declare one or more **actions**
Service can be registered at a **directory-service**

Some of the relationships in Table 4 pertain to agents on a multi-agent system level rather than on a purely internal scope. For instance, `agent-platform` knowledge defines inter-agent relations in terms of the `message-transport-service`, `directory-service`, and `naming-service` used by a group of agents. Here are a few concrete examples that might appear in FIPA relations:

- **HTTP** is a type of **FIPA-transport-protocol**
- **KQML** is a **Agent-Communication-Language**
- **KIF** is a **content-language**
- **XML-Encoding** is an **Encoding** into **XML**

- **Transfer-of-Learning** is a **Service**
- **Soar** is an **Agent**
- Transfer-of-Learning may be provided by Soar

The second FIPA document mentioned in this section builds on the information in the first document. It also relies heavily on analysis of existing system and proposed mechanisms (protocols, services, etc) to create powerful design abstractions (FIPA 1999b). In doing this, the resulting FIPA architecture intends to separate behavior from mechanism. The way FIPA accomplishes this is similar to Dr. Kendall's approach. An agent*policy* is produced as an abstract specification that captures how an agent element or agent interacts with other elements in an agent system. This is done by formally defining constraint expressions (preconditions, post-conditions, and invariants) that capture the interaction. An example of one policy would be an agent requiring that all messages the agent exchanges with other agents must be encrypted. Since policies are independent of agent mechanisms (instantiations), policies can be reused and changed dynamically.

2.3.7 Others

Though eliciting agent domain characteristics like architectures, roles, properties, attributes, components, and relations appears satisfactory to some researchers, it is not to all. For instance, Wooldridge and Jennings point out that languages could and do exist for specifying agents as well as building them (Wooldridge 1995). These same authors claim that agent theories to guide use of languages according to theoretic principles are also needed. They refer to a set of such theories beginning with Dennett's *intentional systems theory*. Therefore language and theory selection are two other possible areas for classifying agent the domain.

This concludes the review of key agent domain research. Several aspects or dimensions of the agent domain have been introduced. Some of these dimensions have been classified or

further partitioned. Despite the breadth of the provided information, more exists. For example, Nwana moves beyond agent classification by property and architecture by suggesting classification by roles and goals (Nwana 1996).

2.4 Summary

In summary, a knowledge base system is an abstraction of a database system intended to contain not just data, but any class of knowledge. It would even be possible, following the approach of Sernadas and Sernadas, to have a knowledge base system of knowledge bases. Not only can knowledge base systems manage virtually any domain of knowledge, but they can also take on very different structures. At a high level, if behavioral knowledge and complex relationships are to be captured, then an OO or possibly a relational knowledge base system is needed. On the other hand, if the knowledge is effectively modeled by a directed graph or tree structure, then a network or hierarchal knowledge base system will be more efficient.

Options exist for choosing a lower-level structure as well. In fact, there is a wide selection of representations for domain knowledge to take within a given knowledge base system meta-structure. Rules are excellent for well-defined domains of question-and-answer type information (like the medical diagnosis system), but they don't work as neatly with structured knowledge such as representing parts of an automobile and their interrelations. Structured declarative knowledge approaches, in turn, may not be the best choice where flexibility in operations on knowledge is needed. This could be accomplished, alternatively, by taking part of the inference engine's black-box procedural operation and including it as procedural content in the knowledge base, which is exactly what frames and objects can provide. Still other cases exist where a hybrid or meta-representation would be better than any of these pure approaches.

Apart from, and as important as knowledge base structure, is determination of knowledge base content. Much research has been done in the agent domain in extracting knowledge. Some

of the more applicable research was reviewed in this chapter. The following chapter presents a methodology for correlating and building upon that knowledge and incorporating it into a well-structured knowledge base.

III. Methodology

Chapter 2 explained knowledge bases in terms of 1) their content, 2) the structure of the content, and 3) the meta-structure or system for containing and accessing this content. This chapter outlines the methodology for establishing all three of these items for an agent-oriented knowledge base system.

A sensible starting point for a methodology would be an understanding of the knowledge to be stored. Once this knowledge is understood, it can be captured into appropriate representation structures, which can then be organized for storage in a selected knowledge base meta-structure. These three primary steps summarize the Knowledge Base Development Methodology (KBDM) to be followed (Figure 7). Details regarding each step appear in the next few subsections.

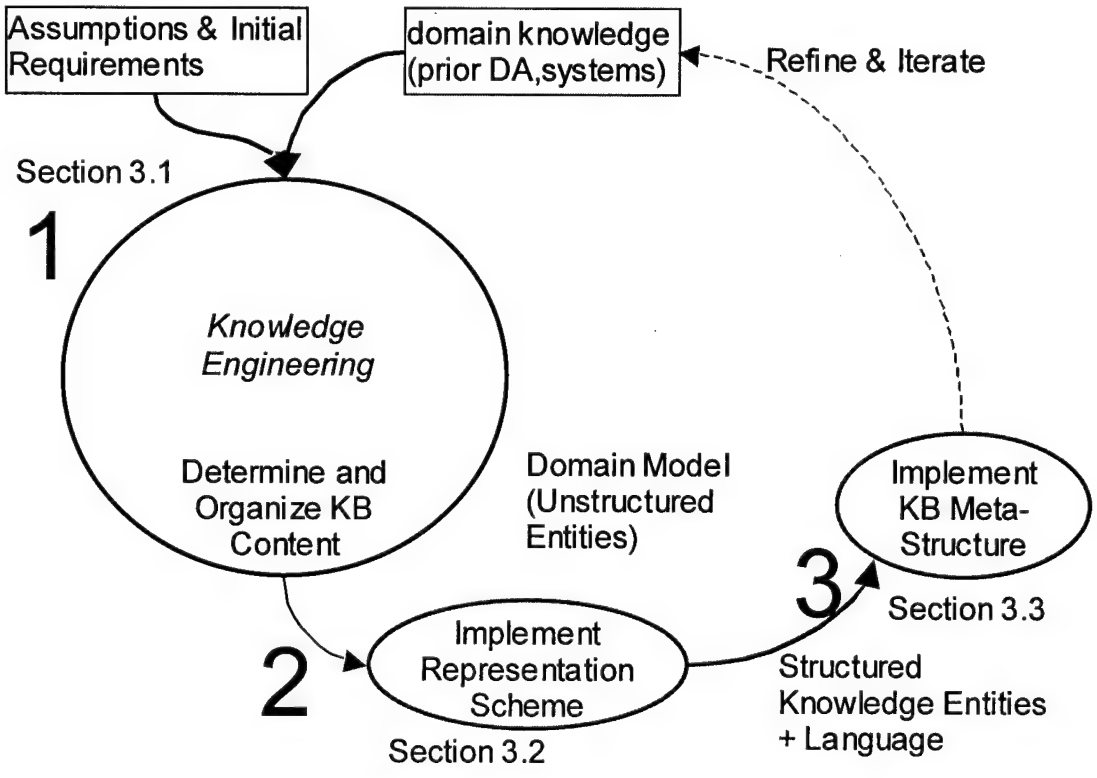


Figure 7: Knowledge Base Development Methodology (KBDM)

3.1 Determine and Organize KB Content

The first step of this general methodology (that of determining the knowledge and organizing it for storage) is known among computer scientists as *knowledge engineering.* Knowledge engineering requires 1) an expert in the domain to provide the knowledge, and 2) a domain analyst that can effectively collect, filter, and organize domain knowledge from one or more such experts. Figure 8 captures the two key stages of knowledge engineering: 1) *Preparing Domain Information* and 2) *Analyzing the Domain.* These sub-stages of the KBDM capture the essentials of preparing the knowledge, analyzing it based on requirements, and creating an organized knowledge structure. As reflected in Figure 8, the resultant knowledge structure would then be mapped to a selected representation scheme, which, in turn, would be stored in a meta-structure for the knowledge base system. The knowledge engineering sub-stages precursory to these latter steps will now be explained further.

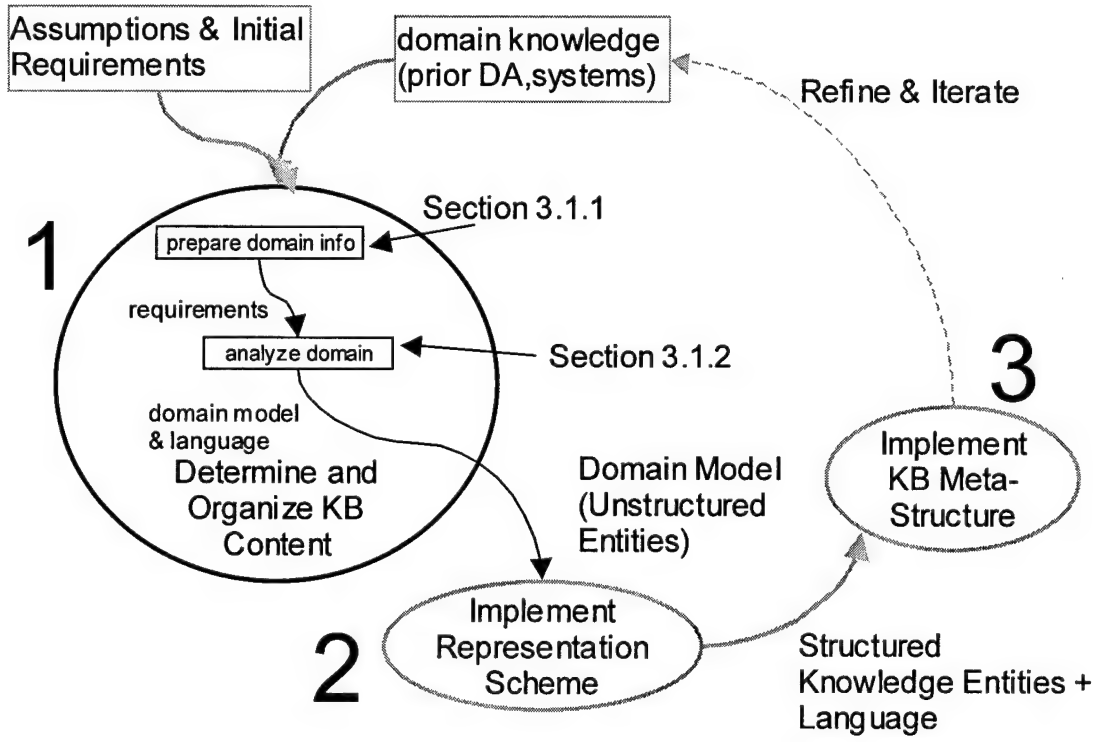


Figure 8: Domain Engineering Process (Warner 1993)

3.1.1 Preparing the Domain

Preparing the domain for analysis is, in short, 1) systematic research and 2) summarizing that research in a *Domain Analysis Requirement Document* (DARD). This step involves extracting what the domain experts view as the essence of the knowledge area as a bounded set of requirements. For example, a mathematics knowledge engineering task may involve questioning mathematicians but would not involve querying veterinarians. It may not even involve gathering calculus knowledge if that is not a requirement (it would be out of bounds). Knowledge can be obtained either by a series of question and answer sessions or by review of the literature. Often a domain has had systems developed for use in it already. In such cases the existing systems themselves can be analyzed to extract domain knowledge. Of particular interest in this stage of KBDM is the collection of prior domain analysis efforts in general, especially those geared towards creating any existing systems. Warner, in his complementary knowledge-based effort, noted that a "knowledge base must [also] support the code generation component of a knowledge-based software engineering system by providing a library of reusable software specifications of implementations" (Warner 1993). Since code generation is a goal of agentTool, both specification and implementation level agent knowledge needs to be collected in preparing the domain information for later organization and conversion to a reusable form by the domain analyst. This requirement for specification and implementation analysis is an example of something that would appear in the DARD. It also leads to the suggested addition of meta-knowledge as a focus of the knowledge engineer. The relationships between knowledge elements such as implementations and specifications form this meta-knowledge. Collected and generalized meta-knowledge should appear in the DARD when available.

In addition to providing collected knowledge, meta-knowledge, and referenced existing systems, certain other information should appear in the KBDM DARD. Warner lists 14 specific items as particularly useful in effecting domain analysis (Warner 1993). Among these are:

- Definition of "domain" (and/or the particular domain)
- Determination of problems in the domain
- Permanence of domain analysis results
- Focus of analysis
- Approach to reuse
- Primary product of domain development
- Relation to the software development process
- Purpose and nature of domain models

Not all of these items are clear as to exactly what they represent, but it is the knowledge engineer's responsibility to clarify and justify whichever of these are selected for inclusion in the DARD for the analyst's benefit. However, the KBDM requires as a minimum 1) the definition of the domain to be analyzed (e.g., agent domain), 2) a description of the relationship the domain model has to the parent software development tool (e.g., agentTool), 3) potential analysis problems, and 4) recommendations.

To exemplify exactly what is accomplished in this step, consider the package-shipping domain. A knowledge engineer is tasked to develop a system to handle package shipping. The engineer's requirements bound the package-shipping domain to cover non-international ground and air shipping. He is also given additional guidelines that scope the type of shipping activities

that must be captured (pick-up, payment, delivery, etc). With this information, he first consults inventory and customer service personnel and then examines a couple of programs that handle shipping for other businesses. He is fortunate to discover a prior, though somewhat ad hoc, domain analysis that was used for development of one of those other programs. Next, from the bounds, guidelines, and specific knowledge gathered, the knowledge engineer produces a DARD for the domain analyst to use. This document may suggest a breakdown of shipping domain activities and modules, refer to the existing systems, and provide other relevant shipping knowledge. Included also is a definition of the shipping domain and a description of how a model of it is to be used in the future application. It definitely contains the prior domain analysis results that the knowledge engineer located. However, this document does not contain the final domain analysis (which may need to be more extensive than the included analysis), but it does help guide a successful domain analysis effort.

3.1.2 Domain Analysis (DA)

The core task of a domain analyst is filtering and organizing domain knowledge. Several domain analysis approaches are listed by Warner (i.e., Arango, Neighbors, Iscoe, Kang, McCain, Prieto-Diaz) *(Warner 1993)*. KBDM will combine strengths of each these approaches with contemporary object-oriented principles. For simplicity, this Combined Objectified Domain Analysis Methodology will be called CODAM. There are four basic steps to CODAM as well as one important underlying principle. The principle is that the Unified Modeling Language (UML) will be used for visual representation of the domain model and taxonomies. The steps are:

1) **Identify Abstract Objects**: This step involves looking at collected knowledge and existing systems for determining what objects are in common. This may require decomposition of the collected knowledge to a level where commonalities can be found. For instance, individual attributes and methods may need to be examined.

2) **Identify Abstract Associations and Operations:** As with the previous step, common entities are collected. However, these entities are associations between objects. Operations are common activities performed on objects.

3) **Identify Abstract Relationships:** Relationships differ from association in that they capture structure rather than behavior. In this step specific common relationships are extracted from collected knowledge. These relationships may then be abstracted. In the case of aggregation, the individual parent-child relationships are identified during abstraction.

4) **Perform Classification:** In this final step all the above information is organized into a domain model as well as hierarchies capturing inheritance.

These component steps of CODAM also appear in Figure 9.

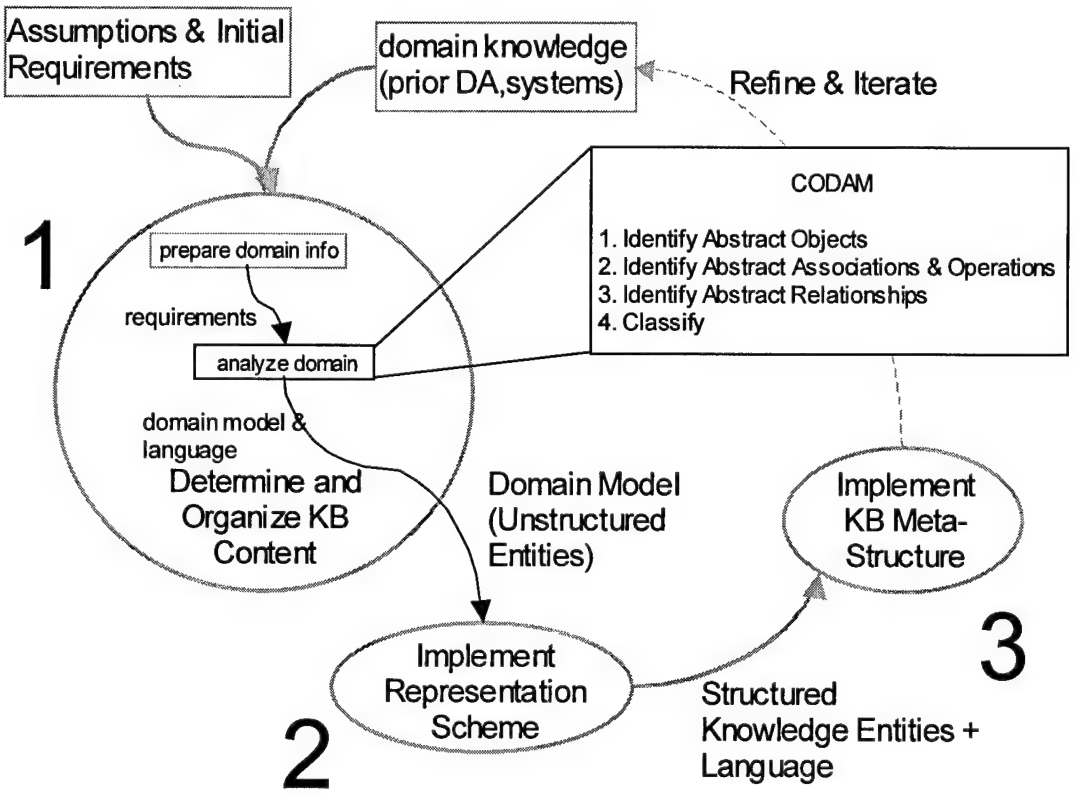


Figure 9: KBDM CODAM sub-Methodology

Earlier, in the *Prepare Domain Knowledge* stage, a package-shipping example was given. The end product of that example was a DA Requirements Document for use by the domain analyst. That example can now be brought one step further. The domain analyst takes that

document and uses it as a guide while following CODAM. The following lists what may occur in the four steps the analyst then follows.

1) **Identify Abstract Objects:** The analyst reviews the prior DA effort for the objects found to be common. He compares the results to other existing systems and generates a final list of common objects. That list may include `sending-customer`, `receiving-customer`, `package`, and `vehicle`.

2) **Identify Abstract Associations and Operations**: At this point, the analyst identifies associations such as: `sending-customer` ***sends*** `package`, `package` ***intended-for*** `receiving-customer`, and `package` ***sent-on*** `vehicle`. He does the same with operations, to include: *`change-name-of`* `customer` and *`assign-vehicle-to`* `package`.

3) **Identify Abstract Relationships**: Relationships may be specific to individual systems such as: `package [10, Dayton-NYC, #104]`, which states that a `package` is composed of a `weight`, `origin-destination`, and `customer-number`. The analyst can take these specific relationships and abstract them. Thus, for instance, the `package` object above abstracts to: `package` ***has-an*** `origin`, `package` ***has-a*** `destination`, `shipment` ***has-a*** `weight`, and `shipment` ***has-a*** `customer-number`.

4) **Perform Classification**: The analyst now creates a taxonomy of objects and other pertinent knowledge. The transport object taxonomy, for example, maintains a `vehicle` at the top level and several subclasses below it (`van`, `semi`, `airplane`, `railroad`, etc). The domain model is also generated to capture associations, relationships, and hopefully aspects of the taxonomies.

The next section examines how the domain model produced here is used by KBDM.

3.2 Implement Representation Structure and Generate Domain Language

The knowledge collected and organized in the domain analysis portion of knowledge engineering resulted in production of a model of the domain. To be usable by an application, however, three things must occur. First is the selection of a scheme for representing all modeled knowledge. Next is the mapping of domain model entities into that representation structure (Figure 10). Last is the generation of a language common to the application system and domain model. Each of these is discussed below.

3.2.1 Select a Scheme

A set of implemented domain objects is the primary product of this stage. The implementation vehicle is a representation scheme. Among the representation schemes available are logic, rules, semantic nets, frames, objects, and hybrids such as CKML and ASTs. In the event that the domain model is complex enough so that no single representation scheme is ideal for all abstractions, multiple representation schemes may be required. The advantages and disadvantages of these schemes were presented earlier, providing criteria for this selection (Section 2.2).

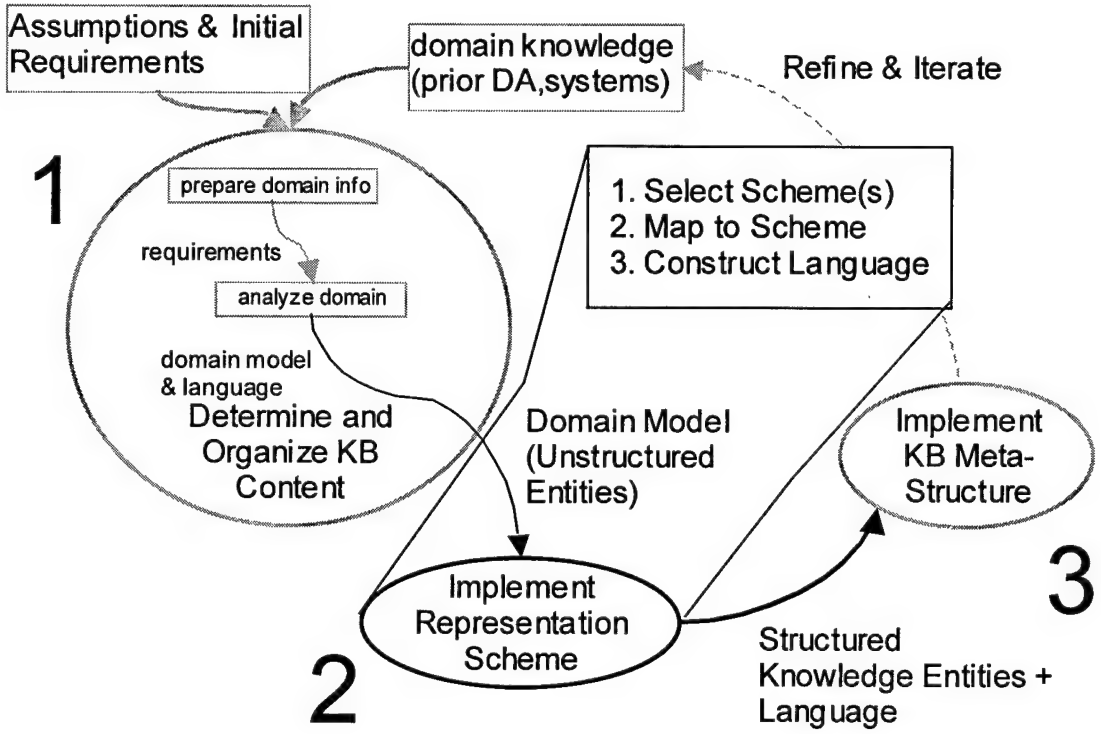


Figure 10: KBDM Stage 2 - Implementing Representation Scheme

3.2.2 Map to a Scheme

Depending on how well organized and consistent the domain model is, mapping the domain model's content into unique representation structures may be either simple or complex. If the domain entities lend themselves well to a standard modular implementation structure then they are simply rewritten in the form that structure compels. When domain analyst Arango referred to moving a reuse infrastructure specification into reuse infrastructure implementation, he was classifying this same type of operation (Arango 1991). Care must be taken to ensure that taxonomic information (knowledge instances) as well as the abstract domain entities and relations are all represented.

3.2.3 Construct Domain Language

The second product of this stage is a domain language. The primary intent of a domain language is to provide a textual syntax (and possibly semantics) for capturing operations involving domain entities. In other words, it provides a grammatical context for an application or user to understand and interact with a domain model. The most straightforward way to create the domain language is to give a unique name to each object, relation, and activity abstraction in the domain model. This was partially done with the package-shipping domain earlier using words such as: `customer`, `send-to`, `origin`, etc. How the resulting language grammar is actually used by the application environment is an issue handled at the knowledge base meta-structure level, which is discussed next.

3.3 Implement Knowledge Base Meta-Structure

Once elemental knowledge structuring is achieved and a domain language specified, the meta-structure for the knowledge base itself is selected and implemented (Figure 11). The result is a complete knowledge base system (KBS). The KBS structure may be strongly dependent on the representation scheme chosen for domain knowledge entities. The domain language also

biases selection of this structure. The reason for the latter is that access methods to individual domain entities are directed through the meta-structure, so the meta-structure will need to absorb that domain grammar into its design in order to correctly perform access functions. In database terms, the domain language becomes a Data Description Language, which is important for database (knowledge base) operation. Section 2.3 correctly suggested then that a database would be the most appropriate meta-structure (due to flexibility, primarily). The selected KBS will need to provide a minimum access method set to include 1) adding new knowledge object, relation, and activity abstractions, 2) removing any of the preceding, or 3) retrieving any of the same for modification or use. Likewise, any of these same operations should be permitted on specific instances of the abstractions. Figure 11 captures this step as well as *Refine & Iterate*, a succeeding step that places the resultant knowledge base system in the position of an existing system at the start of a new KBDM cycle.

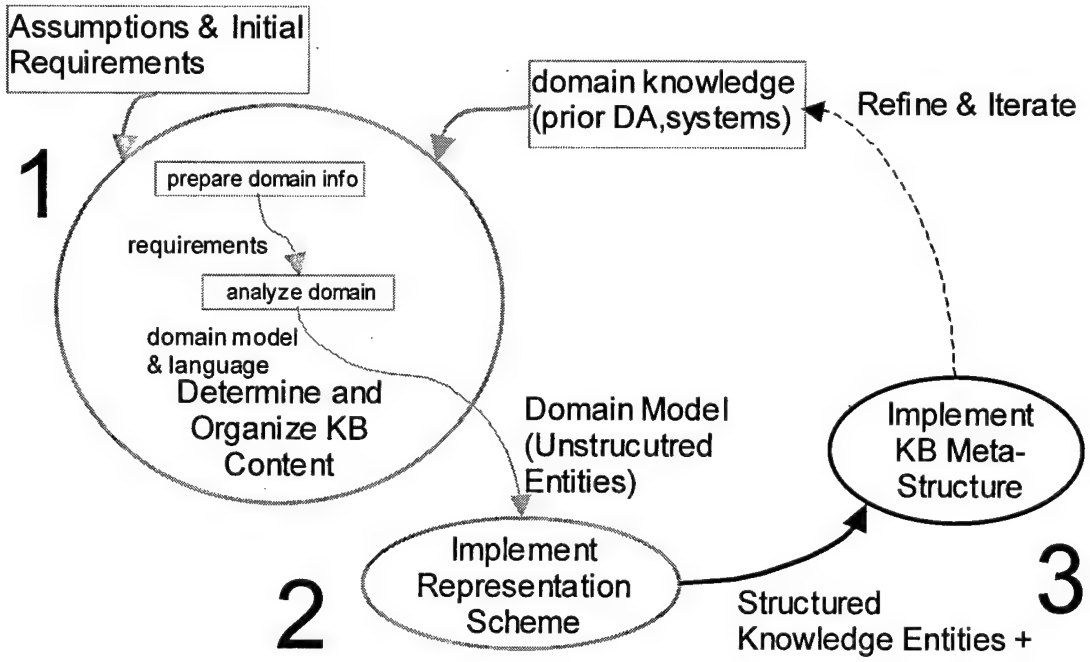


Figure 11: KBDM Stage 3 – Implementing KB Meta-Structure

3.4 Summary

This chapter presented Knowledge Base Development Methodology (KBDM), a methodology for designing a knowledge base scoped to a particular domain and containing persistent, reusable knowledge. The *scoping* for this effort is to be achieved by knowledge engineering of the agent domain. Results of that step take the form of a Domain Analysis Requirements Document (DARD) that contains definitions, assumptions, requirements, recommendations, and other pertinent information to guide domain analysis. Using the DARD as a guide, a domain analyst produces a domain model and taxonomies. Reusable knowledge can then be mapped from that model and those taxonomies into 1) a language and 2) a representation structure. These two products are next combined into a implementation knowledge base system. Access to knowledge objects will be enabled by the incorporation of the domain language into the meta-structure in the form of methods for adding, deleting, finding, and retrieving knowledge. The next chapter will detail the design decisions made in applying KBDM in the agent domain.

IV. Knowledge Support Design

The Knowledge Base Development Methodology (KBDM) discussed in the previous chapter has five products; 1) a Domain Analysis Requirements Document (DARD), 2) a model of the agent system design domain, 3) a selected knowledge representation scheme for the contents of this model, 4) a domain language and grammar, and 5) a knowledge base meta-structure implementation for containing the domain knowledge, integrating the domain language, and supporting the representation scheme. This chapter presents the implementation of these five products resulting from application of the KBDM. Section 4.1 briefly discusses the initial preparation stage (Prepare Domain Knowledge) and introduces the DARD. Section 4.2 then addresses decisions made during domain analysis and presents the domain model. Section 4.3 covers decisions relating to mapping knowledge elements from the domain model into a selected representation scheme and incorporating that scheme into a knowledge base meta-structure. Section 4.4 then concludes the chapter. Although design decisions are made throughout these sections, key decisions will be highlighted due to their particularly significant impact.

4.1 Prepare Domain Knowledge Decisions

In Section 3.1.1 the process of preparing domain knowledge was summarized as being the systematic collection and summarization of domain knowledge in a Domain Analysis Requirements Document. The DARD for this effort (Appendix A) contains:

1) Definition of *domain* and *agent domain*
2) Explanation of the agent domain model's place in the software engineering process
3) General analysis requirements and assumptions
4) List of existing systems that may be useful to the analysis
5) Potential problem areas and recommendations

The domain definition and one requirement that appear in this effort's DARD are reproduced below to add clarity for transitioning to the next stage of the KBDM, *Domain Analysis*.

> Agent Domain Definition: *A collection of knowledge for multiple dimensions/aspects of agent including: concepts, designs, specifications, and implementations of agent systems, agents, and agent components that together completely capture agent information at every level of representation (abstraction) and stage of development.*
>
> DARD Requirement: The domain analysis should support the Multi-agent System Engineering (MaSE) development methodology, by capturing knowledge entities that are used by that approach.

4.2 Domain Analysis

In the past, several efforts have attempted to organize or specify the agent domain. Most of those efforts focused on narrowly scoped portions of the domain. The efforts include (see Section 2.3):

1) Franklin and Graesser's classification of agents by properties and components.

2) University of Michigan's delineation of agent architectures and distinguishing of them by their properties and components.

3) McGill University's UAA, an attempt to create an abstract agent architecture with universally usable components and implementation framework.

4) University of Cincinnati and Stanford's JAFMAS and JATLite communication frameworks and communication specifications.

5) Elizabeth Kendall's analysis and specification of roles, role models, and behaviorally modeled RMIT agent architecture.

6) FIPA's various specifications for standardizing and defining agent relationships, architecture, and constitution.

Of these, FIPA's is exceptional because it examines the agent domain macroscopically and is consorted effort by several agent domain researchers. However, their consensus-driven approach has a drawback: FIPA's standards are yet very much in draft. To build on the strengths

of FIPA and non-FIPA research approaches, this effort examines the agent domain space broadly yet without extensive peer consensus.

The final result of domain analysis is a domain model and a related collection of taxonomies. Figure 12 generalizes the domain model into two primary, though overlapping*, sub-domains: an *agent entity* sub-domain and a *multi-agent system* sub-domain. Both sub-domains contain knowledge at conceptual, design, specification, and implementation levels. One-way arrows on the figure represent relationships that exist between knowledge levels (*mappings*). The larger multi-way arrow represents non-mapping relationships (*associations)* between elements at various knowledge levels in both sub-domains. Specific mappings and associations are discussed later in this chapter. However, descriptions for the four knowledge levels and some facets of each are provided here. Note that each of the identified facets of each knowledge level corresponds to a single class of abstract knowledge object

> ***Conceptual Knowledge Level***: A concept is a general idea. Agent concept facets include *goals* (objectives for the agent or multi-agent system to achieve) and *properties* that characterize behaviors of agents or agent systems. Most agent development starts with generating conceptual knowledge.
>
> ***Design Knowledge Level***: A design is a general pattern or method. Agent designs organize and add meaning to agent concepts. Design knowledge facets include:
>
> > ***Role***: An intended pattern of behavior, responsibilities, and collaborations for an agent class within an overall structure or system. Roles contain generalized *tasks* and a set of *resources*.
> >
> > ***Communication***: A simple model of an interaction between *roles*.
> >
> > ***Role Model***: A collection of *roles* and patterned *communications* between roles.
> >
> > ***Task***: A method that constrains and defines how a *goal* is to be achieved.
> >
> > ***Resource***: Something that a *role* has access to that aids or supports the *role* in its responsibilities.

* The sub-domains overlap because the *agent* knowledge element exists in both sub-domains, linking them.

Specification Knowledge Level: A specification is a detailed and exact description of an agent or agent system. Agent and agent system specification facets include:

> ***Agent***: The key functional entity of the agent domain.
>
> ***Component***: The fundamental functional module of an agent.
>
> ***Architecture***: An abstraction for using a set of *components* together in a particular way. Although an agent always has an architecture, *components* themselves may also have architectures for organizing their sub-components.
>
> ***Communication Framework***: A set of protocols, components, and mechanisms that permit relay of data and knowledge. Java Remote Method Invocation (RMI) and sockets are two common communication frameworks. Framework specifications have close ties with framework implementations.
>
> ***Data Construct***: Analogous to a resource for a role, but for an agent. Data constructs are not necessarily part of an agent, though an agent may have access to them. Components of an agent may utilize this access in order to function.
>
> ***Conversation***: Basically a state-transition table (or finite-state machine) defining interactions between agents.

Implementation Knowledge Level: An implementation is the equivalent of a complete specification though in a compilable and runnable form.

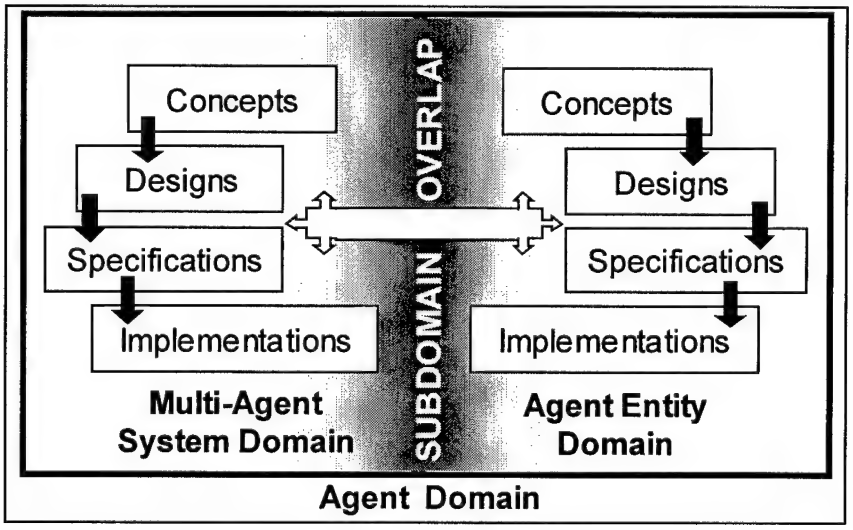


Figure 12: Generalized View of Agent Domain

By inserting these and other knowledge facets into the generalized domain model, and by then applying the KBDM/CODAM principle of using UML in modeling the domain, the model shown in Figure 13 was produced*. Four shades of objects are used to differentiate between facets in different knowledge levels (i.e., implementation facets are darker than conceptual facets).

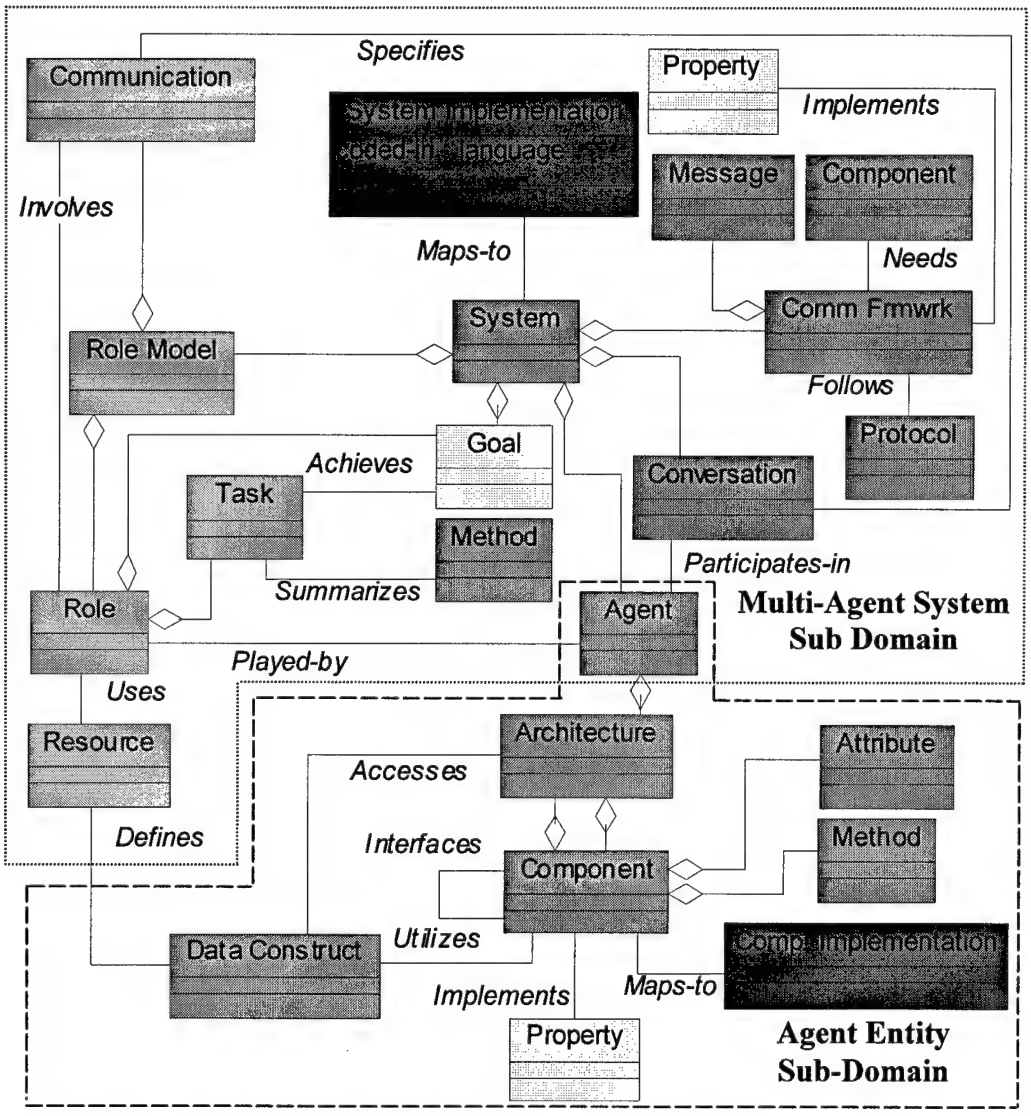


Figure 13: UML Model of Agent Domain

* *Method* and *Property* appear twice strictly to reduce visual clutter.

More information on the knowledge elements (facets), the taxonomies they represent, and element associations appear in the following two subsections. Section 4.2.1 discusses knowledge contained in the agent entity sub-domain while Section 4.2.2 covers multi-agent system sub-domain knowledge.

4.2.1 Agent Entity Sub-Domain Analysis

This portion of the agent domain is depicted in Figure 14. Agents are composed of architectures, components, attributes, and methods. Data constructs and properties are also associated to an agent via its components, as are component implementations. Presenting the origin and relationship of these knowledge elements is the focus of this section.

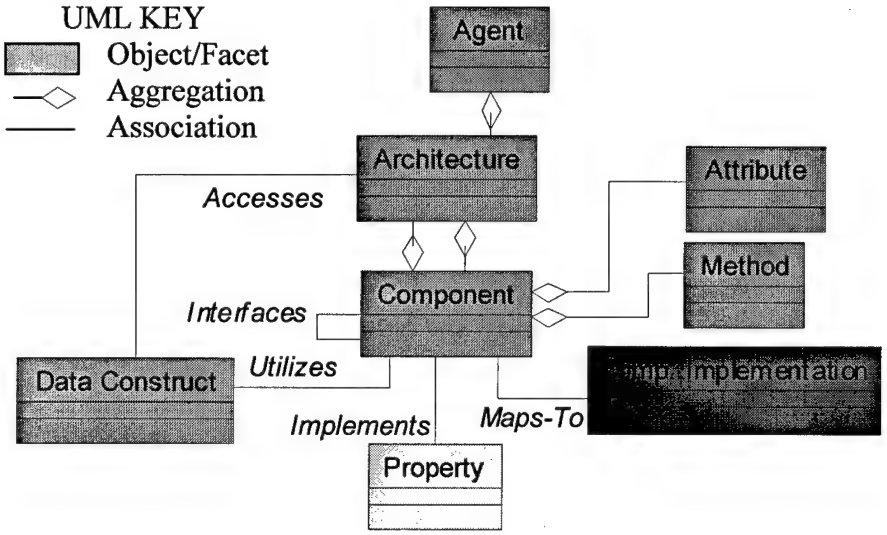


Figure 14: Agent Entity Sub-Domain

Any analysis of the fundamental entity of the agent domain, the agent, is incomplete unless it poses the question "What is the definition of 'agent'?" or at least "Which definition of agent is correct?". The answer to either query is found by analysis of the myriad of circulating definitions of 'agent'; and that answer is that they all define what an agent is, providing that each

definition is considered in context. This is possible because each definition intends to define a specific *class* of agents, though few, if any, describe every class of agent. For example, the general definition of agent introduced in Chapter 2 actually defines just what an *autonomous* agent class is.

> An *autonomous* agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda so as to effect what it senses in the future (Franklin 1996).

Other definitions likewise refer to classes of agents with other properties. Viewing this another way, *properties* provide a means to conceptualize or define agent classes (or multi-agent system classes). Therefore, rather than present various definitions of agenthood, this effort presents the properties that those definitions all use. A property taxonomy appears in Section 4.2.1.2.

Though agents can be organized and identified by their properties, non-conceptual knowledge must exist to provide a foundation for implementing those properties. Analysis of the research summarized in Chapter 2 indicates that agent architectures and components implement properties on behalf of agents owning those components. This is not to say that an agent is a empty shell, but rather it is composed of an *architecture* of interoperable components that realize the properties that are associated with that class of agent. *Components*, like the agents themselves, may have an architecture with encapsulated sub-components. These two elements (components and architectures) fall into the specification knowledge level, as do data constructs. *Data constructs* represent resources available to an agent, and thus to its components. For instance, a `virus-definition` data construct could be one that a `virus-checker` component of an agent, with some `security` architecture, would require to function. Just as properties form the core of agent conceptual knowledge, components, architectures, and data constructs form the core of agent specification knowledge. Taxonomies of three of these (but not data constructs) will be introduced in Section 4.2.1.2.

Abstract architecture, property, data construct, and component objects have identifiable attributes such as names and descriptions. In addition, data construct attributes may include a set of data fields detailing the internals of the construct. For example, `virus-definition` may have data fields for `virus-name`, `discovery-date`, `size`, `affected-OS`. To keep consistent with the CODAM principle of using UML wherever possible, all attributes that are not reusable will be represented as UML attributes. Reusable knowledge elements like the attribute and method object parts of the component entity, however, will be attached using UML aggregation. This decision is reworded below and is apparent in the UML models that appear in this chapter.

> **Key Design Decision #1**: Aggregation will used whenever the lower level object has significant potential for reuse. This means that components will be found in aggregations but not as attributes of architectures. It a means that names and descriptions will appear as attributes. Objects defined in the multi-agent system analysis stage will follow this same guide. At the implementation stage, of course, aggregated objects may be stored separately or as part of the aggregation.

4.2.1.1 Agent-Entity Association Knowledge

Before considering specific instances of conceptual and specification knowledge sub-types, associative knowledge needs to be presented. In Figure 12 associative knowledge takes the form of 1) a set of one-way arrows that map from one knowledge level to its neighbor and 2) a multi-way arrow that maps within and between the agent sub-domains. The mapping arrows serve the purpose of transferring knowledge from a lower level to a higher level (e.g., concepts to designs, designs to specifications, and specifications to implementations). Figure 14 shows that one of these mappings is Maps-to. *Maps-to* is used to link a specified component to the actual software code implementation of that component. Other mappings will be discussed in Section 4.2.2.

The multi-way arrow shown on Figure 12 represents a large set of non-mapping associations, to include 1) *Interfaces,* 2) *Utilizes*, 3) *Accesses*, and 4) *Implements*. The *Interfaces*

association exists between two components in an architecture that need to coordinate their behavior in some form. One use of this association might be in capturing the fact that a `virus-checker` component needs to address a `knowledge-store` component in order to look up stored `virus-definitions`. This differs from the *Utilizes* association, which reflects that a component requires an interface with a data construct directly. *Accesses* is related to *Utilizes* in that it specifies that an agent has access to a given data construct. An agent with such access can have its components utilize them. For defining the final association, *Implements*, captures the relation between a property and the component that realizes that property (see Key Design Decision #2).

> **Key Design Decision #2**: Properties may be implemented by either individual components or sets that compose an architecture. To simplify this in the domain model, any architecture that *implements* a property will be assignable to components and that component will implement the property on behalf of the architecture.

More detailed behavior than shown by these associations is captured by individual component *methods (e.g.*, `find_smallest_num(array)` is a method that takes an array and finds the smallest number in it).

4.2.1.2 Agent Entity Knowledge Taxonomies

A key aspect of the domain not reflected by the domain model is the *depth* of the domain space for each knowledge facet (e.g. property, component, etc.*). Just as the original knowledge levels broke down into these facets, those facets break down into taxonomies. Each taxonomy has a static top-level partitioning under which numerous instances and subclasses are organized. Having a taxonomical classification will be important when the domain model is implemented in a knowledge base because it will allow for easier organization of reusable knowledge, accelerating both storage and retrieval. Due to time limitations, only a small subset of existing

* Tables 1-3 provide most of this information.

systems are classified in the taxonomies presented by this effort. This can be seen in the following sub-sections as those taxonomies are presented.

4.2.1.2.1 Property Classification Taxonomy

Classifying properties is challenging for two reasons: first, properties are not tangible, and second, properties are tightly related to capabilities. Though capabilities have not been mentioned to this point, they were assumed in the University of Michigan study discussed in Section 2.2. They differ from pure properties in that capabilities identify what agents can *do* (ability) while properties identify what agents *are* (traits). Rather than promote semantic confusion, this iteration of KBDM will consider properties and capabilities together in a single hierarchy. Figure 15 shows several key properties/capabilities in hierarchal form. Though the definitions for the properties appearing in this figure were presented in Tables 1 and 2 in Chapter 2, descriptions of key top-level properties appear here for convenience.

> ***Mobile***- Able to transport itself from one hardware unit to another (e.g. between two networked computers).
>
> ***Communicative***- Able to communicate with other agents or a user.
>
> ***Reactive***- Responds in a timely fashion to changes in the environment.
>
> ***Learning***- Changes its behavior dynamically based on acquired knowledge, whether that knowledge comes by experience or by instruction.
>
> ***Planning***- Able to sequence actions to reach a desired goal. Although there are multiple types of planning, most require a memory/storage facility.
>
> ***Reasoning***- Able to consider alternatives and make weighted decisions.

4.2.1.2.2 Architecture Classification Taxonomy

Prior agent architecture classification attempts influenced the design of this effort's domain taxonomies, including the property taxonomy just discussed (Arriola 1994; Robinson 2000). Whereas the Michigan study classified architecture by platform or research team and

produced a lengthy list, Robinson lists just five architectures (reactive, planning, knowledge-Base, Belief-Desire-Intention (BDI), and user-defined). Both of these approaches are used in this effort by 1) identifying a small set of architectural styles and 2) classifying each architecture instance under one of those abstractions. The decision to format domain taxonomies in this way was suggested in Section 4.2.1.2 and appears as Key Design Decision #3.

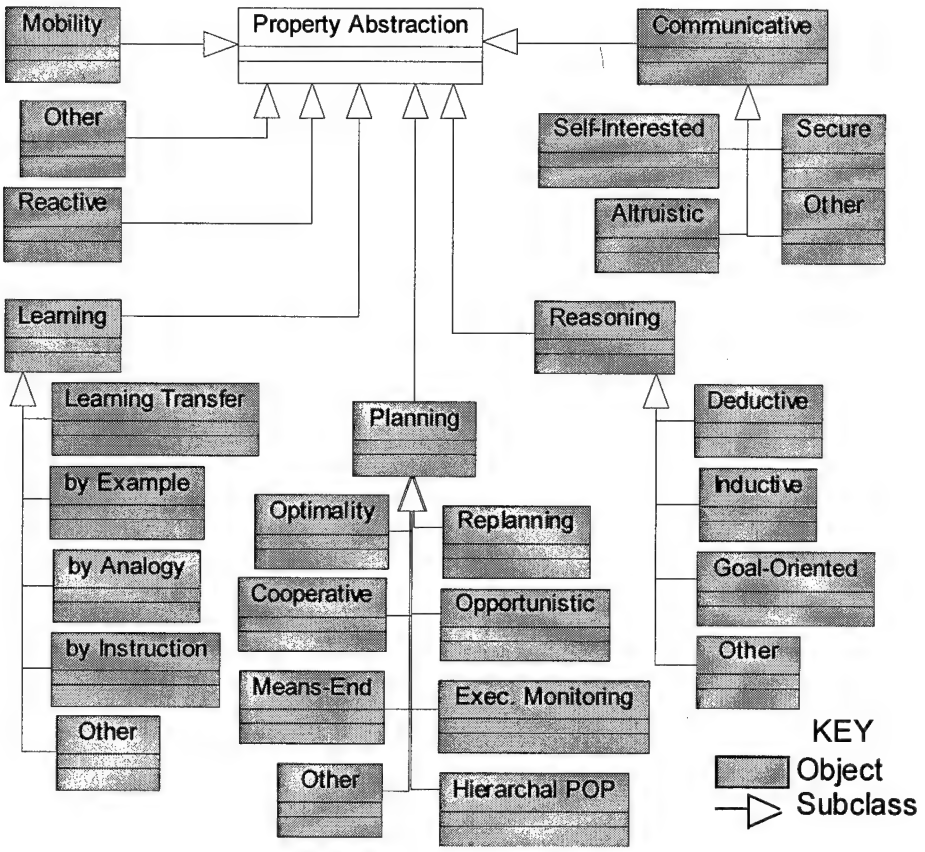


Figure 15: Property Abstraction Hierarchy

Key Design Decision #3: The agent architecture sub-domain is partitioned into two levels of abstraction. Highest is a set of abstract architectural styles. Below these styles are platform-dependent architectures conforming to constraints imposed by the styles. Below this level may exist yet more specific architecture instances. In the event that a platform architecture will not map to one of the fundamental styles, a new *composite* style may be added to the hierarchy. Other domain taxonomies follow a similar bi-level format.

Figure 16 presents a set of architectural styles similar to Robinson's but with two noteworthy differences. First, Robinson's *planning* style is captured by the CODAM Pure-Deliberative architectural style (*Deliberative* for short). More complex deliberative systems that are based on human-like reasoning models are captured by the *Pragmatic* architectural style. Second, Robinson's *user-defined* style is replaced with the generic *Composite* style, which identifies architectures that mix aspects of the other three categories (deliberative, reactive, and pragmatic, and composite). *Reactive* architectures, as the name would suggest, are ones that directly support implementation of the reactive agent property. *Classic* architectures are those that use an inferencing style like that found in classic rule-based expert systems (Robinson 2000).

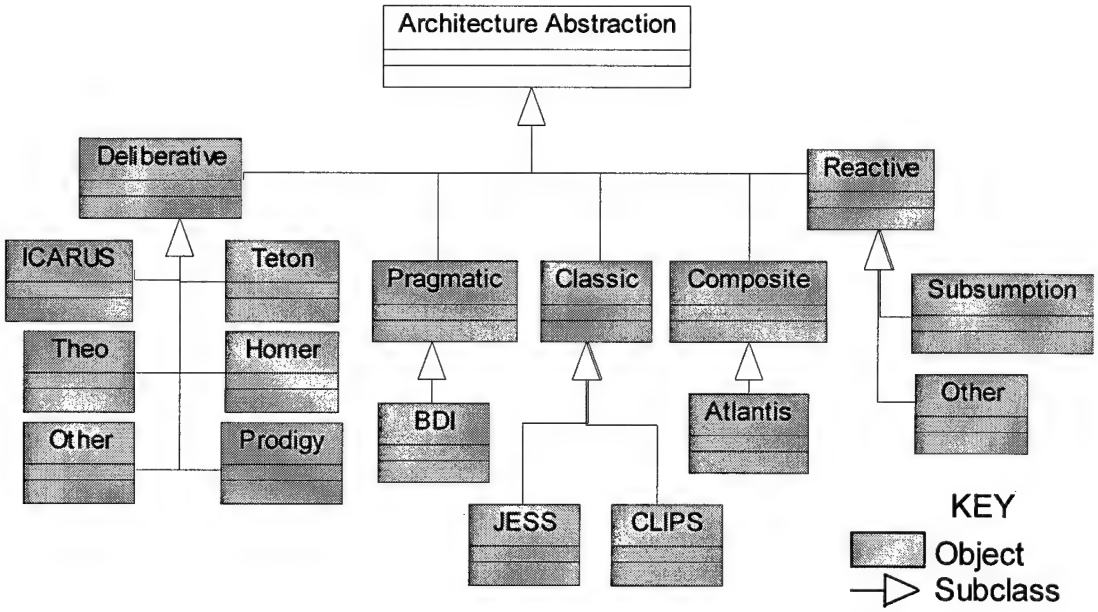


Figure 16: Architecture Abstraction Hierarchy

4.2.1.2.3 Component Classification Taxonomy

Figure 17 depicts the five key categories of agent components as well as several specific subclasses of each. The selection of the six areas follows logical functional partitions between the behaviors of components of existing agent systems (Ndumu 1999). *Planner* components are those that constitute and directly support planning while *Knowledge* components permit

organization, storage, and retrieval of data. *Execution* and *Service* components are somewhat related, the former capturing general control and execution structures and the latter more task-specific operating components, which includes interface components to applications, communications systems, data constructs, and the environment in general. *Model* components model the environment an agent works in so that agents can maintain a local representation of their world. Any other components not captured in one of these five areas would fall under the *Other* classification. A handful of definitions for component types appear below.

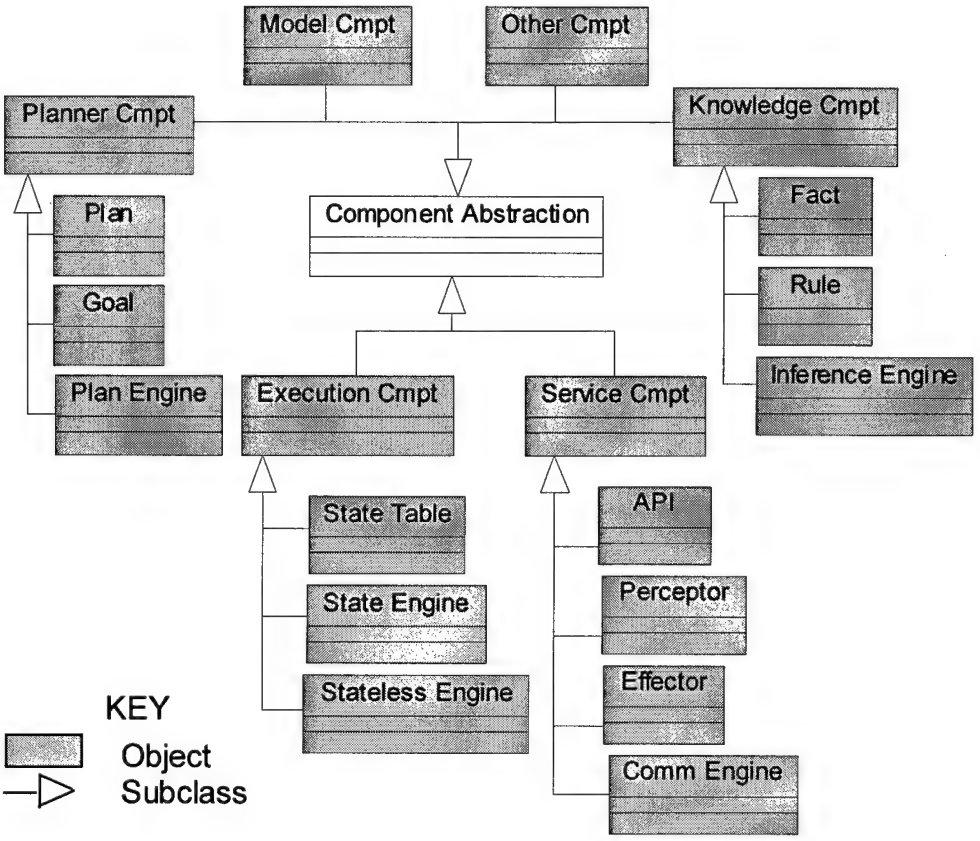


Figure 17: Component Abstraction Hierarchy

Communication Engine- A key service component directly supporting frameworks (see Section 4.2.2).

API- An application programming interface (API) models how an agent can access and interact with a particular resource. Common resources are people and databases.

Inference Engine- Component maintaining access to a knowledge storage structure and performing inferencing functions on knowledge in than structure, such as chaining rules.

Perceptor- A component allowing an agent to sense environmental changes. Perceptors are key in implementing reactive and autonomous agents.

Plan Engine- Used by an agent to assemble plans dynamically and reason.

State Engine- The component of an agent governing its execution if the agent is state-based.

Though not obvious from the taxonomy, components really have two taxonomies. The taxonomies are identical in composition though one contains component specifications and the other component implementations. Implementations have been referenced as the end products of deploying an agent or multi-agent system specification. Each component specification-implementation pair is linked by a mapping relation, which was discussed earlier.

This concludes analysis of the agent entity sub-domain of the agent domain space. The high-level 'agent' entity focused on to this point will be one of many entities considered at the system level in the following section.

4.2.2 Multi-Agent System Analysis

This section provides an analysis of the multi-agent system domain, and thus deals with what agents compose (systems) rather than what composes the agents. Specifically, a *multi-agent system* can be defined as:

> A software program designed using agents as the main programming element, and thus inheriting all the advantages (e.g., distribution) and disadvantages (e.g., narrow scope) of that construct.

Figure 18 provides an overview of the multi-agent sub-domain. The conceptual, design, specification, implementation, and association knowledge shown (Figure 18) are discussed below.

Like agents, multi-agent systems are modeled conceptually by *properties*. Multi-agent systems also have goals, though agents generally achieve those goals. Unlike agent properties, properties in multi-agent systems are not directly implemented by components and architectures. Instead, system properties are emerge from agent interactions at the design level. These interactions then have ties to specification and implementation facets. The interactions are captured in *roles*, *role models*, and inter-role *communications*. Multi-agent system *communications frameworks* do what architectures did for agents; they implement the properties and permit inter-role communications to occur.

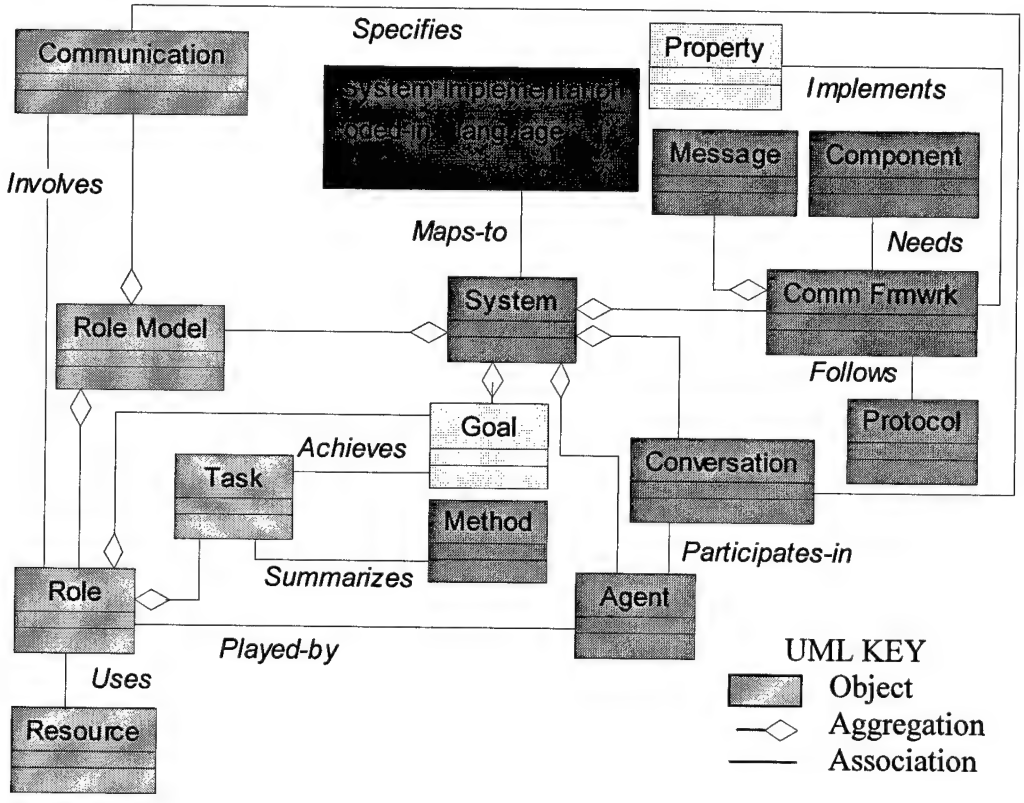


Figure 18: Multi-agent System Sub-Domain

For this effort, all communication frameworks require the existence of four other specification types: agents, messages, conversations, and protocols. Though the first three do not

actually compose a framework, they must be present in a system along with a framework in order for it to be effective. Agents have been described already, *messages* are the objects passed to relay information between agents, and *conversations* are refined descriptions of inter-agent interactions governing message sending and other activities. Because of the way MaSE (the agentTool development methodology) uses conversations, a design decision was required for domain modeling of them.

> **Key Design Decision #4**: Because MaSE models conversations as dialogues, the domain model must minimally do the same. However, the capability to capture broadcast conversation should be possible without significant change in the domain model.

The fourth specification entity related to frameworks, *protocols*, specifies the constraints placed on the basic communications using that framework. Frameworks may also require certain components to exist in all agents in a system using that framework, so that those agents can interact. One of these components, the *communications engine,* appeared in Section 4.2.1.2's component taxonomy. Each unique framework would have a unique communication engine (i.e. a Java RMI framework would have an RMI communications engine and follow RMI protocols). The conversations that occur between agents over a framework are basically finite-state machines, complete with states, transitions, and guards.

Three other noteworthy top-level knowledge elements are the *task, resource,* and *system implementation*. A task is a simple design structure that encapsulates a goal, *resources* needed for achieving that goal, and possibly some applicable constraints. Tasks are collected in role entities whose responsibilities match the task function. A resource represents something that aids or supports a role in achieving its tasks. Some tasks may require multiple cooperating roles. Related tasks may be assigned to a single role or to a group of roles in the same role model. The role model, then, would also contain interaction knowledge (communications) applicable to those

roles. The final knowledge element, the system implementation entity, contains code for a system just as component implementations contained code for component specifications.

4.2.2.1 Multi-agent System Association Knowledge

Like agents, multi-agent system knowledge elements have both mapping relationships and non-mapping relationships between them. The *Maps-to* mapping exists between a system specification and the implementation that realizes it. *Played-by* is a mapping relation that assigns a *role* design to an *agent* specification (e.g., an agent plays a role). *Summarizes* maps a task to a specific component method. Finally, the *Defines* mapping links a design resource to a data construct specification of that resource. These mappings appear in Figures 13 and 18.

Several non-mapping associations also exist in the multi-agent system domain: 1) *Follows*, 2) *Implements*, 3) *Achieves*, 4) *Uses*, 5) *Involves*, and 6) *Participates-in*. These associations are identified in Table 5.

Table 5: Multi-agent System Associations

Association Name	Association Description Template
Follows	A Framework *Follows* a Protocol
Implements	A Framework *Implements* a Property
Achieves	A Task *Achieves* a Goal
Uses	A Role *Uses* a Resource
Involves	A Communication *Involves* a Role
Participates-in	Agent *Participates-in* a Conversation

4.2.2.2 Multi-agent System Knowledge Taxonomies

As with the agent entity sub-domain model, the multi-agent system domain model does not show the full depth of the domain space. Taxonomies fill this need here as well. However, the incredible diversity of possible tasks, goals, resources, and conversations makes it unwieldy to produce their hierarchies at this time (this same situation arose with data constructs). However,

framework and role model taxonomies are presented in Figures 19 and 20. Short descriptions of some specific frameworks and role models are listed as well.

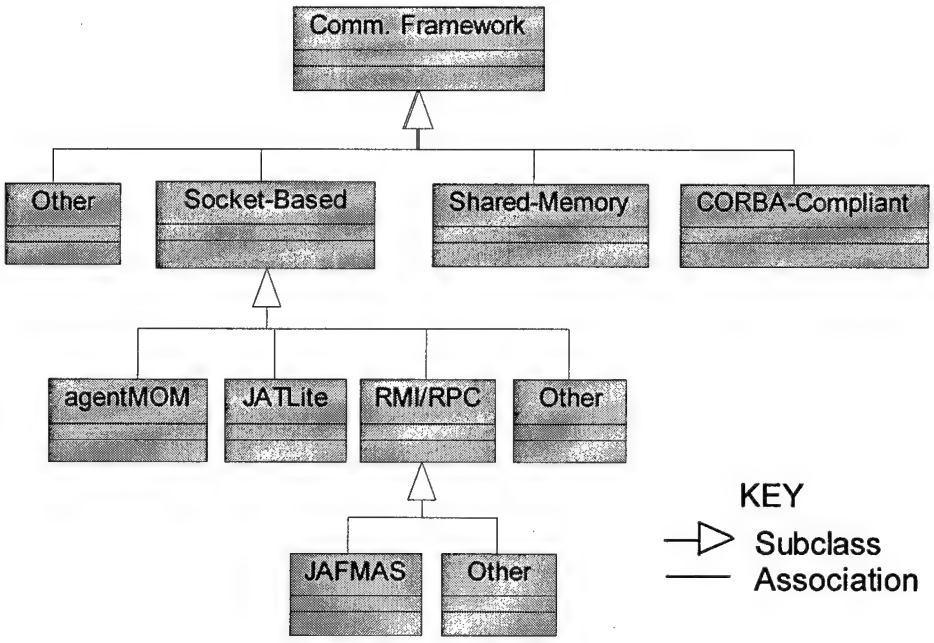


Figure 19: Agent System Framework Hierarchy

Frameworks:

Socket-Based – These frameworks use a two-way communication link between two programs running on the network, where each end is called a *socket*. Most socket connections are client-server and use TCP protocol, though multi-casting and broadcasting sockets expand beyond this to allow for groups of programs to share a single socket network connection. The underlying protocols in such those systems are usually UDP, RAMP, or similar multicast-capable protocol. JATLite is a particular incarnation of a socket-based framework (Chapter 2).

agentMOM- Message-Oriented Middleware for agents is a standard set of objects: conversation, message, etc. that allows socket-based message exchange. The knowledge base of this effort uses this to communicate with agentTool.

Shared-Memory- A framework where a common storage resource (usually main memory) is used for message exchange. One agent generally leaves a message with a recipient tag in the shared area, and at some unpredictable future time the attended recipient checks the memory and recovers the message. Similar to a blackboard architecture.

RMI/RPC- Remote Method Invocation in Java; Remote Procedure Call in other languages. RMI is slightly more powerful (and uses RMI sockets). Both allow distributed objects to call methods in each other without explicit declaration of sockets by a user.

CORBA-Compliant- A framework that follows CORBA standards. Slightly more flexible than RPC because a handle on an object, not just an object method is available remotely.

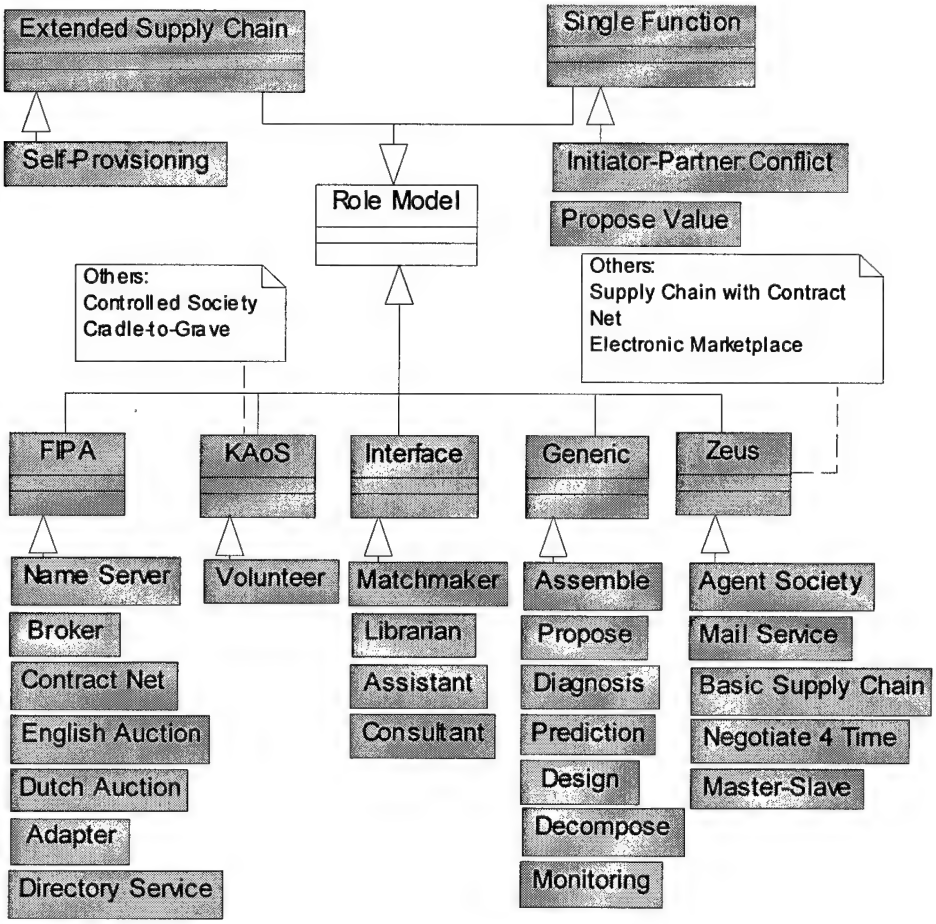


Figure 20: Role Model Taxonomy (Kendall 1998a)

Role Models:

Contract Net – Based on the contract net protocol. It is a task allocation paradigm where the allocation is realized by a negotiation process between agent roles. A manager (role) with tasks to be executed, contacts contractors (roles) that may be able to execute those tasks.

Dutch and English Auctions- These are closely related role models. Both have several bidder roles and one seller role. In the Dutch auction, the seller requests bits at an ideal price (could be time units for execution), and gradually lowers the price until the bid is met. English auction usually has the seller request all bids and selects the highest as winner.

NameServer- This role model has a name server role and several client roles. A client will register with the name server what the client's capabilities are. The name server then can respond to queries from other clients who are looking for a client with certain abilities.

4.2.3 Summary

This concludes the domain analysis portion of the KBDM. In Section 4.2.1 the agent entity domain analysis was discussed. This provided a foundation for the successive multi-agent system domain analysis in Section 4.2.2. Combining the collected sub-domain knowledge and organizing it according to the design decisions made thus far provides the overall domain model as shown in Figure 13. Following KBDM, the next stages of knowledge base development transform that domain model into a useable knowledge base implementation.

4.3 Knowledge Base Design

In this section a representation scheme is presented and then incorporated into a meta-structure for managing the represented domain knowledge. Subsection 4.3.1 considers various scheme options, and provides arguments for selecting a hybrid. The selected hybrid representation consists of two parts: an object model for the agentTool execution environment and a text-based extensible Markup Language (XML) model for storage and transfer that is called the Multi-Agent Markup Language (MAML). The text model doubles as a representation scheme and domain language, and is described in subsection 4.3.2. The storage meta-structure, called the Agent Random-Access Meta-Structure (ARAMS), is described in Section 4.3.3. Figure 21 illustrates how the domain model, object model scheme, agentTool, MAML model scheme, and ARAMS interact.

4.3.1 Find Candidate Schemes

Selection of a representation scheme for a domain should never be arbitrary. Even when domain entities seem to lend themselves to a given scheme, consideration must be given to the behavioral and relational content that affects those entities. For instance, though the semantic net representation scheme introduced in Section 2.1 can capture associations, it may fail to disallow an unnecessary association. Beginning without bias towards one scheme or another is the ideal approach. Logic, rules, semantic nets, frames, objects, and common hybrids are then competing on equal grounds. Even with such equality, examination of the domain model quickly rules out logic and rule-based schemes for their inability to encapsulate structural knowledge. Semantic nets also fail to adequately capture relationship and association knowledge, though they do so better than logic and rules (see example in Figure 2). This leaves only frames, objects, and hybrids as true candidates.

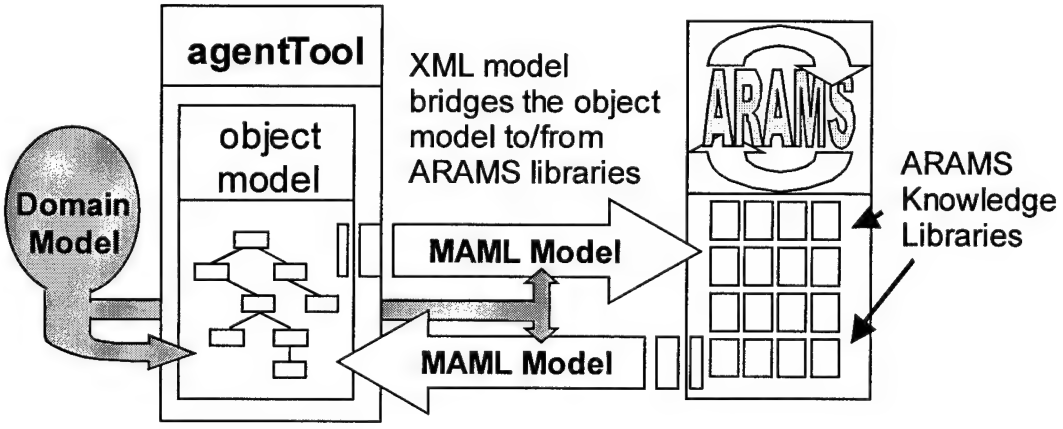


Figure 21: Relationship Between Products of the Effort

For this effort, the prior assumption that selection of a knowledge representation scheme would start without bias must be thrown out due to agentTool's Java requirement (see Chapter 1). Despite the bias toward schemes promoted by Java, selection of an effective scheme is not hampered since Java permits and thoroughly supports object representation. Considering that

frames are fundamentally objects without procedural encapsulation, frames can be modeled in Java if needed. The choice remaining is then between objects and some hybrid scheme such as CKML or the AST.

In agentTool, agent systems, agents, and conversations are modeled as Java objects. Figure 22 shows the object model of the agentTool agent system design space (as of November 1999). Though the structure only shows the generic objects (like the KBDM domain model), each of the objects can be instantiated to any of several object instances. For example, the *ATconversation* object can be instantiated to a *RegistrationConversation*.

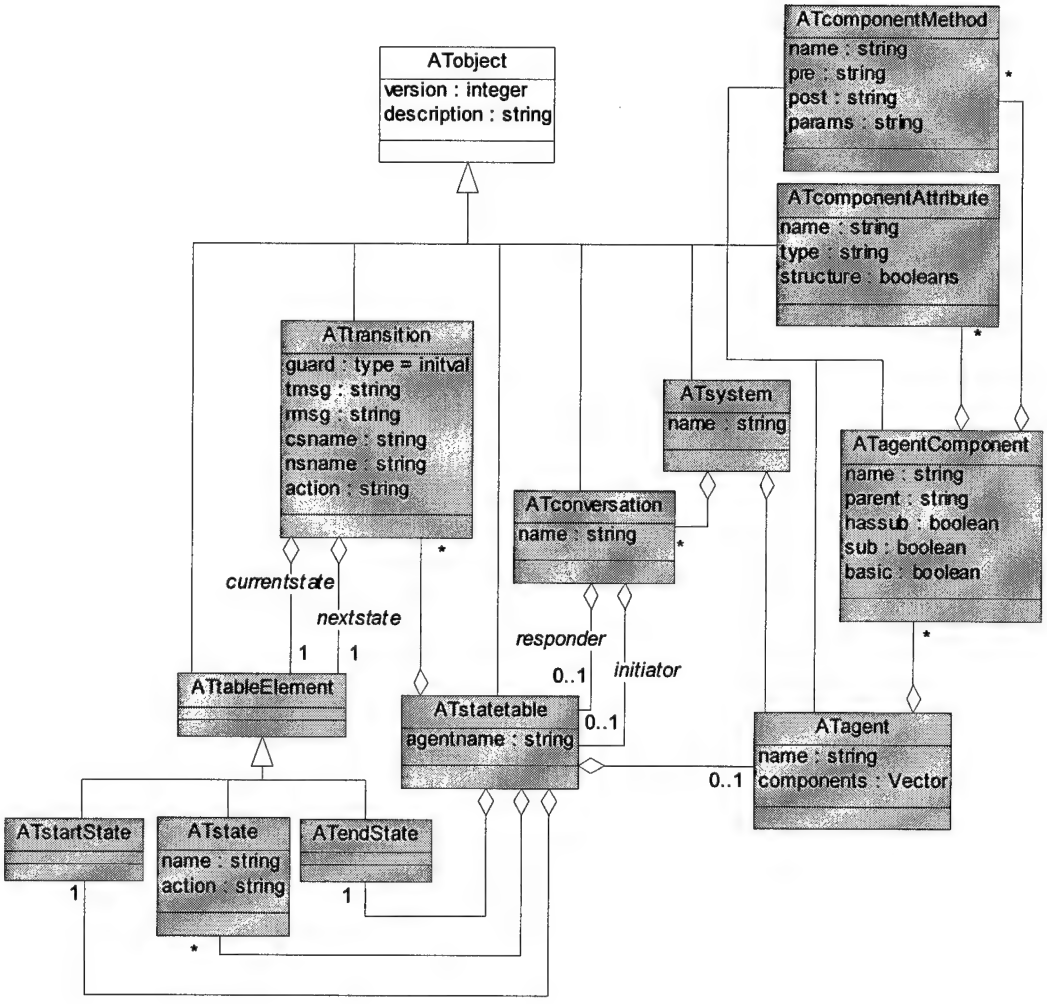


Figure 22: Java Object Model used in agentTool (November 1999)

Though objects originally seemed to provide a flexible and effective representation scheme for the KBDM domain model, this changed during testing of the KBDM's knowledge base when stored agentTool object instances became irretrievable because they could not map to newer versions of their parent object types. This problem is analogous to re-keying the locks on a car - the older version of keys become useless. Although complex Java coding could have provided a partial work-around, a more elegant and flexible approach was chosen. This approach is captured in the following design decision.

> **Key Design Decision #5**: KBDM and agentTool will use two representation schemes. The first is an object scheme for modeling agent domain objects in the dynamic environment of agentTool. The second, the Multi-Agent Markup Language (MAML), will be used for static, persistent storage of agent domain knowledge.

4.3.2 Multi-Agent Markup Language (MAML)

The Extensible Markup Language (XML) provides a flexible but powerful solution to the need for a secondary representation scheme. Using XML not only eliminated the object-verison problem, but it reduced the size of represented objects by orders of magnitude. The Multi-Agent Markup Language (MAML) is the application of XML in this effort. KBDM's MAML offers three advantages over alternative forms. First, MAML captures the domain grammar. Second, MAML represents all design objects in compact and persistent form, its primarily aim. Lastly, MAML is language and platform-independent, meaning that it is possible for agentTool to interact via MAML with an entire virtual suite of Java and non-Java agent design tools.

MAML consists of 1) a set of linked Document Type Declarations (DTDs), 2) encapsulated MAML production methods and object-model constructors, 3) a text parsing utility, and 4) a process for applying each of these. Section 4.3.2.1 introduces the first three of these items and how they integrate into the agentTool domain model. Section 4.3.2.2 relates how these elements apply to the process for creating, storing, and retrieving persistent, reusable knowledge

facets. Section 4.3.2.3 finishes this section by discussing how the MAML representation scheme may be extended beyond design object to the implementation level.

4.3.2.1 MAML Integration

Though the object model used in agentTool is dynamic, changing constantly as improvements are made to capturing agent design information, certain parts of the model are fairly static. The single most static component of the model is *ATobject*, the abstract parent of *ATsystem*, *ATagent*, *ATconversation*, etc. This fact influenced key design decision #6.

> **Key Design Decision #6**: Because correspondence must be maintained between what is captured in MAML and what is captured in the object model, MAML methods and parameters will be included in the abstract *ATobject*. Additional concrete extensions and applications of these methods will be made in *ATobject* subclasses as needed.

Implementing this decision led to these additions to *ATobject*:

- *MAML* [parameter]: This string holds the MAML representation of an object.
- *MAMLheader* [parameter]: This string holds header information in the MAML. Use of this information will be discussed later.
- *EncodeMAML()* [abstract method]: This method is defined by each subclass with the purpose of translating object content into a MAML representation to store in *MAML*.
- *ATobject(string)* [constructor]: Constructor receives a string (e.g., MAML string) as input
- *Parseline(string)* [method]: This method parses a MAML string into an array of elements using significant MAML characters as delimiters (<, >, and "). The array is returned. Usually called by subclasses in their *ATObject(string)* constructors.
- *GetMAML()* [method]: Returns the *MAML* parameter.
- *GetFullMAML()* [method]: Returns the *MAML* parameter prepended by a *MAMLheader*, making a self-contatined MAML document readable by XML-focused tools.

- *Extract(string[], int, int)* [method]: Receives an array of string elements and two array index integers, will return a substring of appended array elements from between the two indexes.

- *FindNextInstanceOf(string, string[], int)* [method]: Receives a string, an array of strings, and an index integer. Searches for the string in the array, starting at the index, and returns the index of the first match.

- *FindLastInstanceOf(string, string[], in, int)* [method]: Receives a string, an array of strings, and two array indexes. Searches for the last appearance of the search string within the array, and returns the index of that appearance.

- *FindMatch(string, string[], int)* [method]: Receives a string, an array of strings, and a start index to the array. Searches for a correct match to the given string and returns the array index of the match. A correct match is the appropriate MAML end tag. This method is only needed for components and architectures since they can contain each other in multiple layers.

All of these methods and parameters exist for the sole purpose of encapsulating MAML functionality at the lowest level possible in the object model. The key to these eleven items is the MAML parameter, which holds a string that completely characterizes its parent object instance. Though this is incredibly efficient in both storing and transmitting domain knowledge, this efficiency comes at a price. That price translates to a set of rigorous formalisms that are collected in MAML DTDs and direct how a valid MAML string can be composed. Appendix C contains the MAML DTDs for the primary entities of the agent domain model/object model. These DTDs are used in creating both the *encodeMAML()* methods and the constructors that serve to compliment those methods (one maps from the object to MAML and the other maps from MAML back to the object).

4.3.2.2 MAML Application Process

Though the DTDs are important for formalizing MAML grammar, the encapsulated production and construction methods perform all the work of mapping to and from the Java design objects and persistent MAML entities. Figure 23 shows the process of creating a MAML

entity from a Java object and then re-creating the original domain object from this entity. The seven steps in this process are:

1) A defined (or partially defined) Java domain object is selected. This may be a specific *ATsystem*, for instance

2) There are several sub steps to this:
   2.1 - The *encodeMAML()* method of the selected object is called. This method parses the object attributes into the MAML grammar, building a structured MAML text string as it progresses.

   2.2 - When aggregate objects are encountered, their own *encodeMAML()* methods are called.

   2.3 – The *getMAML()* methods are called to retrieve the generated MAML text from the aggregate objects.

   2.4 - MAML text strings from aggregate objects are then appended to the parent object's MAML text string

3) The system calls *getMAML()* on the selected object (the *ATsystem*, for instance) and redirects the retrieved string into persistent storage.

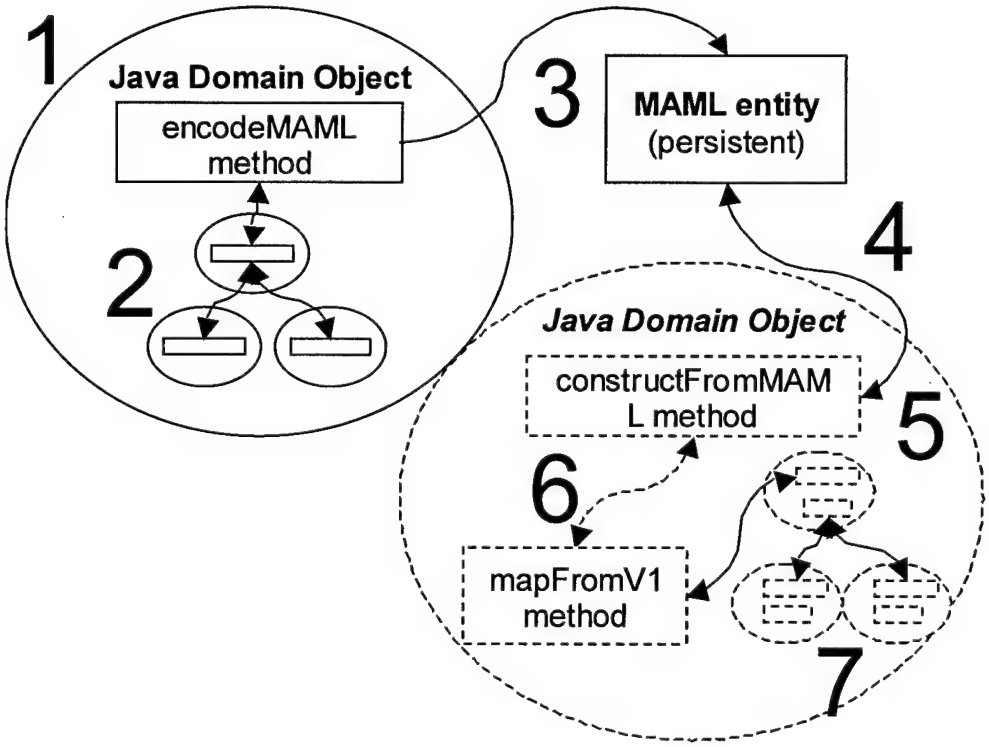


Figure 23:Process for utilizing the MAML representation scheme

4) A stored MAML entity string is retrieved from persistent storage and a new (empty) instance of the appropriate Java object class is created.

5) The MAML string is sent to the new object as a parameter of the *constructFromMAML()* constructor method.

6) The constructor determines the version of the MAML grammar by examination of a field at the start of the string. A method for populating the object (in its current model) from the MAML grammar version is chosen. For instance, if the version of the MAML string was [1] (meaning that its parent object model was of that version), then a method for building the current object (maybe version [3]) from that older version is called.

7) If grammar is encountered for aggregate objects, the version method creates an empty object of the appropriate aggregate type and recursively repeats steps five through seven until the entire original object structure is recreated.

4.3.2.3 MAML at Implementation Level

At this point, the knowledge representation scheme for most of the domain model has been introduced. Although domain design elements such as frameworks have not been captured in agentTool's object model, there is nothing inherent in them to prevent their creation in that model or in the MAML representation scheme. However, consideration has yet to be given to implementation objects and the mappings from the specifications to them. Since agentTool does not address implementation level knowledge at this point, KBDM has unlimited flexibility in developing a storage solution. The only constraint, in reality, is that the form of implementation objects themselves has to be compatible with agentTool's general approach to design. The following design decision was made to assure this compatibility.

> **Key Design Decision #7**: Because implementations will, in the future, be in languages other than Java, and because certain code-level details may need to be mapped from the specifications to those corresponding implementations, implementation objects need to be captured as accessible and reusable source code objects rather than an alternative form.

What this decision means is that a MAML *wrapper* may be placed around a source code object so that that object may then be represented in the same general form as the design objects, a MAML string. In fact, wrapping may be just a start. MAML tags (like HTML tags) may embed the source code object as well as wrap it. This would provide anchors in the code for insertion of specification-level parameters. For example, suppose that an agent specification captures the need for a knowledge base component containing rules. Figure 24a shows a partial component object for this *rulecontainer*. The source code for one specific *rulecontainer* and the associated rule take the form of MAML-wrapped Java objects (Figure 24b). It is notable that there are also MAML tags imbedded in the code to identify distinct objects and static fields (italicized). These anchors may be either removed or replaced by actual rule strings captured in the specification. To support this, a mechanism for parsing the MAML implementation string and operating on it will be needed, though a small variation to *parseline()* may suffice for this.

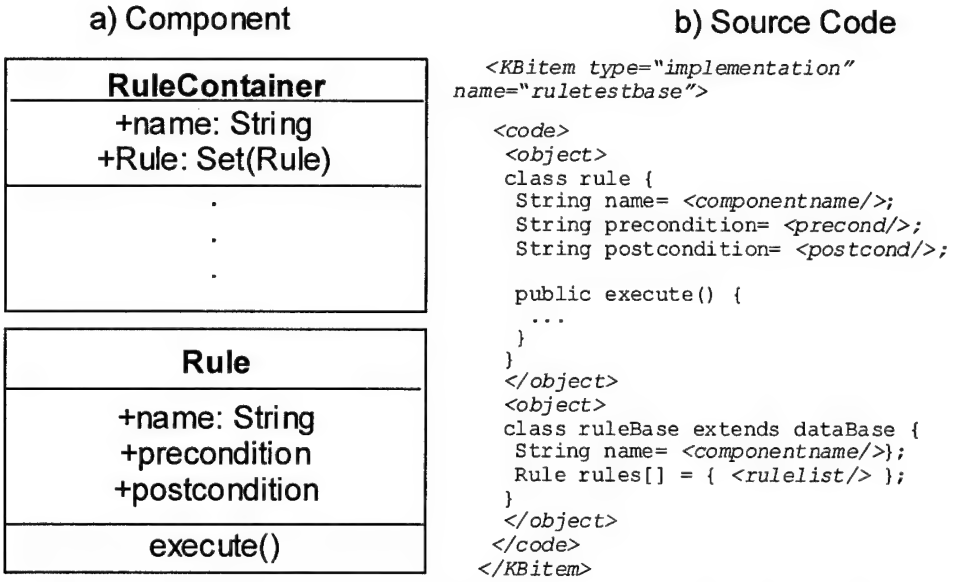


Figure 24:Example Design Object and One MAML-wrapped Implementation

The mappings from the MAML specification objects to MAML-permeated implementation objects may also be captured in the MAML. For this to work, a new mapping

object needs to be created for every implementation stored. Each of these mappings, as with other MAML representations, must conform to a certain subset of the MAML grammar. Application of that grammar is shown in Figure 25, where the *Maps-to* association is defined. This nomenclature's similarity to Figure 12's associations is not coincidental. Indeed, the maps-to association is exactly what is needed for capture and MAML is flexible enough to provide it. Because of this ability, MAML can be used to capture mappings and associations in the agent domain. Such information may be useful as meta-knowledge for more quickly locating an object that meets desired constraints. For example, if a user wants to know what planner components support cooperative planning, it would be time consuming to load every planner design, look at their respective properties, and then look at each implementation for each of specifications that have the desired property. A more logical approach would be to include significant associations and relationships (such as the component *implements* property association) as independent represented knowledge entities that reference other entities (Figure 25b).

This completes discussion of the MAML knowledge representation scheme for the knowledge gathered by the KBDM domain analysis.

```
<relationship type="maps-to">
  <implementation name="ruletestbase">
  </implementation>
  <specification>
  <name>"rulecontainer"</name>
  </specification>
</relationship>
```

a) Maps-to Association

```
<relationship type="implements">
  <component key="jointplanner">
  </component>
  <property key="cooperative">
  </property>
</relationship>
```

b) Implements Association

Figure 25: MAML Representation of Two Associations

4.3.3 Meta-structure Implementation

The final stage of the KBDM requires selection or design of a meta-structure for the knowledge captured by the chosen static representation scheme. An effective meta-structure must do more than simply contain the represented knowledge; it must provide organization and access at a higher level than addressed by the knowledge representation scheme itself. Section 2.3 argued that a database system is the answer to this need. Several classes of database were suggested in that same section, but one with low overhead and significant extensibility is what is needed here.

The decision made in this effort was to use a low-level database with its model being neither strictly OO, relational, network, or hierarchal. Not only does this eliminate costs associated with commercial model databases but, more-importantly, it allows for a simpler, more-tailored solution. Java's built-in support for file access methods makes design of such a low-level structure an even better choice (in addition to providing implicit compatibility withagentTool, which also is designed in Java). The particular Java constructs that provided the most use in designing this database are *Collection* and *RandomAccessFile*. These two structures gave representation ability at the internal schema level and mapping ability to bring knowledge at that level up to the conceptual and external schema levels. The internal schema is essentially a file structure while the conceptual and external schemas appeared in the forms of MAML and the agentTool object model, respectively. The following constraints were met in developing this knowledge base system (schemas, constructs, etc):

1) A common interface for passing knowledge elements was provided.

2) The system did not require that the number of records be known at creation time. This is because the knowledge base was to be manually populated over time.

3) Because of #2, there needed to be a mechanism for dynamically increasing the size of the knowledge structure in long-term storage.

4) Stored knowledge elements were uniquely identifiable.

5) The system provided a means for saving, locating, deleting, and retrieving unique elements or groups of elements.

6) Since the number of stored elements may grow quite large, access methods that function independently of content cardinality were desirable.

In this effort, an existing RandomAccessFile-based database structure was modified in order to extends its abilities (Hamner 1999). The modified implementation is called the Agent Random-Access Meta-Structure (ARAMS).

4.3.3.1 ARAMS

There are two levels of the ARAMS. The first borrows heavily from Hamner's work and deals primarily with general file access. The file type chosen is the Java RandomAccessFile, a persistent storage data structure that has the capability of 1) being sized explicitly, 2) supporting the seek() method, and 3) using DataInput() and DataOutput() interfaces. The significance of these interfaces will be discussed briefly later on. Details of the general implementation of this structure are presented in Section 4.3.3.2. Section 4.3.3.3 captures modification to this basic construct to make it 1) operable with agentTool, and 2) able to contain and organize the MAML scheme objects.

4.3.3.2 ARAMS Foundation

The ARAMS uses a set of record files that act as libraries for different knowledge classes. Figure 26 illustrates the general internal composition that each of these library/record files adheres to. The content and purpose of the three regions illustrated in the figure (internal schema information) are described below.

File Headers Region: There are two pieces of information in this short section of the file. The first is a variable containing the count of the number of stored elements in the file while the second contains a *data start pointer* to the start of

the Records Data Region. Together, these provide a measure the length of the Index Region.

Index Region: In this section are recorded attribute-value pairs, as in a frame representation scheme. Each attribute is a unique key that accompanies a single record header for a record in the Record Data Region (the value). The header contains a pointer to the record itself, a number that measures the space allotted for that record, and a number indicating how much of that space is actually used. Index pairs can be dynamically removed or added as corresponding records are removed or added.

Record Data Region: This constitutes the entire space in the file after the end of the index region. Raw elements (serialize objects) are stored here. Elements can be dynamically removed and added.

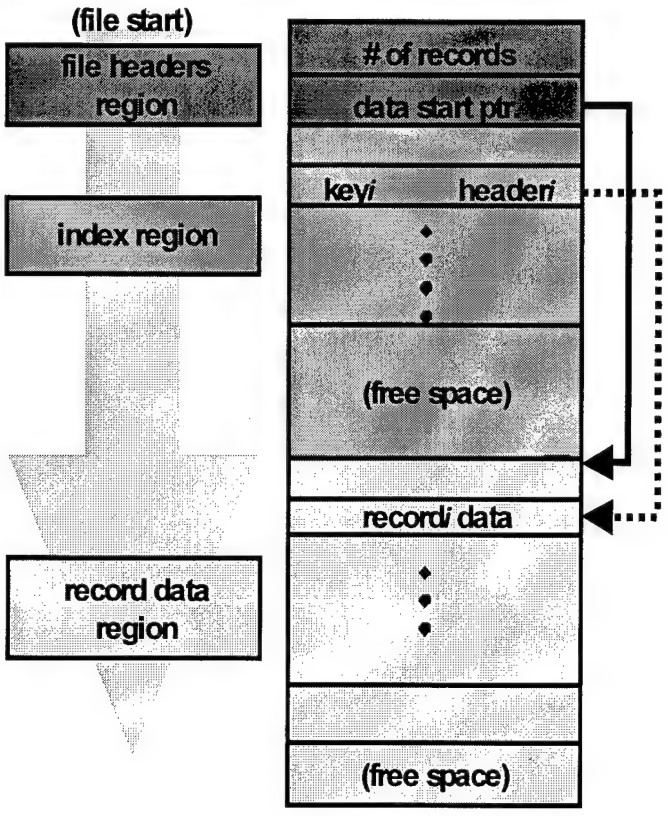


Figure 26: Format for Record File (Hamner 1999)

Several operations are supported by the basic ARAMS architecture. These include *insert*, *read*, *update*, *delete*, and *ensure capacity*. These operations perform the functions that would be expected, with the last one allowing for the index region to expand into the *Record Data Region* if needed (the first record would be moved of course). Earlier, DataInput() and DataOutput()

functions were mentioned. These functions allow information to be written into the record structure without concern regarding the hardware platform used, which is useful in a distributed heterogeneous environment. Yet another operation streams incoming data to an array which can be stored much more quickly. This is better than streaming data directly to the file structure, since that would prevent reads from occurring for an interval, causing a waste of valuable execution time. In addition to these operations, several other tailored functions have been integrated in order to provide special access by agentTool. Section 4.3.3.3 discusses these functions and other aspects of the knowledge base interface to agentTool.

4.3.3.3 ARAMS Extension

In the previous section the underlying file structure of ARAMS was presented. For the purposes of creating a knowledge base for the agent domain, several instances of these file structures will be needed. This is a design decision.

> **Key Design Decision #8**: Because the agent domain knowledge falls into natural partitions, or classes of knowledge, the meta-structure of the knowledge base will be structured along those same partitions.

These file structures contain distinct portions of the agent domain that have been represented in MAML as libraries (Conversations, Architectures, Systems, Roles, Agents, Resources, Data Constructs, Frameworks, Associations, Component Implementations, and Component Specs).

It became obvious early on in this effort, that the knowledge base produced by KBDM would be unique among agentTool components because of *sharing*. Several instances of the agentTool interface may want to extract or add knowledge to the knowledge base simultaneously. Since independent knowledge bases for each such instance would quickly lead to loss of coherence of the platform knowledge base, something was needed to prevent this. The most reasonable solution was to allow the knowledge base system to be a single persistent application in a distributed environment. Ironically, this solution describes an agent.

Key Design Decision #9: Because the prototype knowledge base for this effort needs to be available on-demand to various users in the most current form, it will reside in a distributed environment and communicate with agentTool using agent-oriented communications principles.

This decision led to development of: 1) an *administrator agent* with access to all knowledge libraries, 2) a *connection agent* that is used by agentTool to contact and coordinate with the administrator agent, 3) a set of *conversations* constraining that coordination, 4) communication components for the two agent types that allow them to reach each other over a network, and 5) GUIs for the connection agents to allow users to guide the access process (within the imposed constraints). Figure 27 expands upon Figure 21 with a detailed schematic of how these various elements interact. The Java Collections mentioned in the previous section have an effect at this level. Collections fill the need for the untying of data structure at the storage access level from data structure at the manipulation level. The benefits of being able to do this are many, though the most significant is that they allow for information stored in one data structures to be moved transparently into another structure, or for the original data structure code to be replaced without effecting code elsewhere in the system. This is not a fantastic scenario. Data structures that are highly effective for small amounts of content in the internal schema of ARAMS can become ridiculously slow as the content grows orders of magnitude in size, or when that content is mapped to the conceptual and external schemas (MAML and object models). Using Collections to permit schema evolution was the final key design decision in the KBDM.

Key Design Decision #10: To allow ARAMS to meet the extensibility and flexibility requirements of this effort, Java Collections are used where possible in accessing and passing data structure content.

This completes review of the selection and design of a knowledge base meta-structure for KBDM. This also concludes discussion of implementation of the KBDM as a whole.

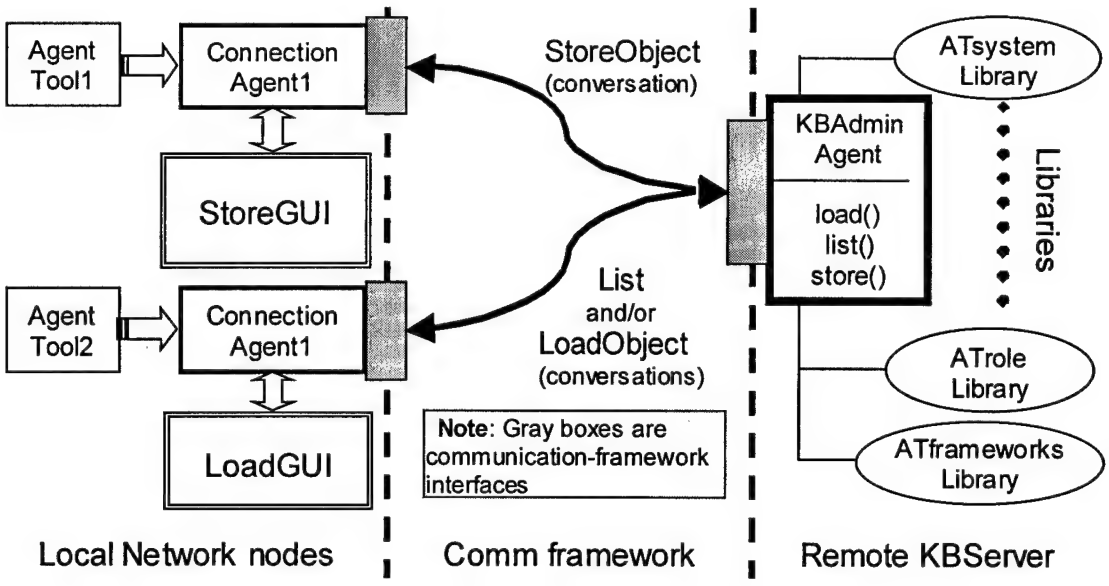


Figure 27: Agent-Oriented agentTool-KB Interface

4.4 Summary

At the opening of this chapter the central goal of implementing the Knowledge Base Development Methodology (KBDM) was presented. The five objectives of that methodology were then outlined as:

1) Completion of a DARD by the Domain Knowledge Engineer
2) Creation of a model of the agent system design domain.
3) Selection of knowledge representation schemes for the contents of this model.
4) Specification of a domain language and grammar (another representation structure).
5) Design and development of a knowledge base meta-structure (KBS) for containing the domain knowledge, integrating the domain language, and supporting the representation scheme.

These products were generated at appropriate points in following KBDM (Appendix B). Section 4.1 laid the foundation of how KBDM could proceed by collecting agent domain knowledge and summarizing the results in a Domain Analysis Requirements Document. From that document, a complete domain model containing associations, objects, and relationships was

produced in Section 4.2. Taxonomies containing specific populations represented by the model's object abstractions were also introduced in that section. Section 4.3 took these results and defined both dynamic and static representation schemes for this collected and organized knowledge. The described dynamic scheme is the agentTool object model while the static scheme is called the Multi-Agent Meta-Language (MAML). MAML doubled as the domain language for KBDM. Finally, but in Section 4.3, a meta-structure called the Agent Random-Access Meta-Structure (ARAMS) was introduced. This structure satisfied the final KBDM requirement by capturing for persistent storage the MAML-represented domain knowledge. It also provided access to that knowledge through an agent-oriented interface to agentTool.

V. Demonstration

The last chapter presented the design of the ARAMS knowledge base produced by applying the KBDM. This chapter presents the results of several tests performed on agentTool-ARAMS in 1) storing agent knowledge (Section 5.2), 2) locating agent knowledge (Section 5.3), and 3) retrieving stored knowledge (Section 5.3). Though the ARAMS administrator agent supports deletion and modification tasks, agentTool does not yet support these. Section 5.1 introduces some key features of agentTool as background to understanding. Section 5.5 summarizes the testing results.

5.1 Understanding the agentTool Interface

Wood's companion effort entails how agentTool's interface is tailored to support a specific agent system development methodology (Wood 2000). Extensions to that interface have been made to support most ARAMS functions. Figure 28 shows the agentTool GUI as it would appear to a user developing a particular agent system. Note the drop-down menu that supports storage/retrieval is visible in the upper-left corner of the GUI. The center and right of the GUI is predominately occupied by the default design window, in which appear several connected boxes representing agent classes and the conversations between them.

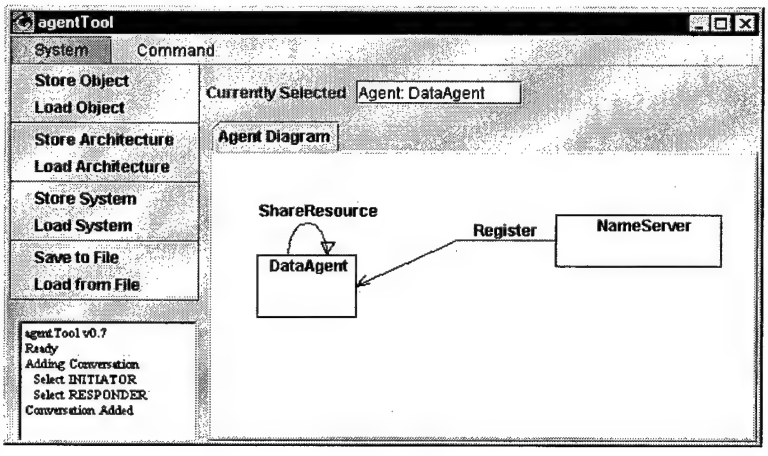


Figure 28: AgentTool GUI

5.2 Storing Knowledge in ARAMS

As any agent system is developed, a user may want to store either the entire system or any part of it for later use. Four agentTool menu options support storage: *Store System, Store Architecture, Store Object*, and *Save to File*. The first three of these operate identically but handle storage to ARAMS of systems, architectures, and elements of these two separately (process generalized in steps 1-2 shown in Figure 22). For an example, the conversation shown in Figure 28 (and expanded in Figure 29) would be stored as follows:

1) User selects the *Register* conversation by clicking on it with the mouse pointer.

2) User selects the *Store Object* menu option

3) AgentTool launches a *connectionAgent*, passing it a handle to the currently selected object. The *connectionAgent* immediately establishes a socket connection with the ARAMS administrator agent.

4) The *connectionAgent* launches a *StoreGUI* as illustrated in Figure 30.

5) In the *StoreGUI*, the user enters "RegisterConversation" in the "object type (key)" field and then enters a detailed description of what that conversation does in the "object description" pane. The user then clicks the *Store* button.

6) *StoreGUI* calls *object.setDescription(d)* which sets the selected object's description field to contain what the user typed.

7) *StoreGUI* calls *object.encodeMAML()* is called which starts the cascading process described in Section 4.3.3.2. This effective translated the object model of the conversation into MAML, which is then stored inside that object.

8) *StoreGUI* creates an ATKB_Object consisting of the user's typed identifier, the MAML string just created, and the classname of the selected object (Atconversation in this case).

9) The *CAStoreObject* conversation side is started and the ATKB_Object is passed to it. The conversation builds message that is sent to the ARAMS administrator agent, which starts its half of the same conversation (*KBStoreObject*).

10) The *keyTest* method is called to see if the user's identifier is already being used in the intended library, *cLibrary* (conversation library). If it is in use, an appropriate reply is sent back to the initiating *connectionAgent*, which displays the message in the *StoreGUI*.

11) If the key is new (see step 10) then the *store* method is called. This method extracts the MAML string and classtype from the received *ATKB_Object*. If the key is already in use, the user is notified and prompted to choose another key.

12) The MAML string and accompanying key are stored in the *cLibrary*.

13) An acknowledgment is sent to the *connectionAgent* notifying the user that the store operation was a success.

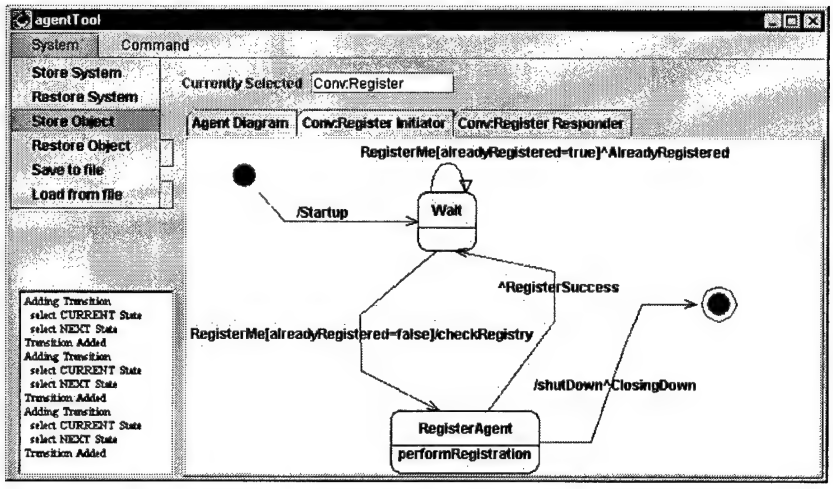


Figure 29: Conversation Design Window

This same basic process is used for storing any agent, agent architecture, component, conversation, or system. Systems may also be stored using the *Save to File* menu. When this option is chosen, the *encodeMAML* method of the current agent system is called and the user is prompted to type a file name for the resulting MAML string. That file is then stored locally to the parent instance of agentTool.

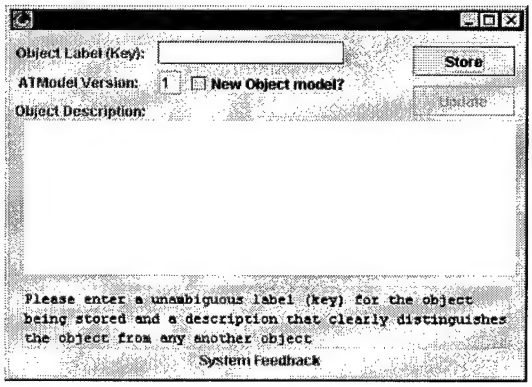


Figure 30: StoreGUI

5.3 Locating and Retrieving Knowledge in ARAMS

The *Restore Object*, *Restore System*, and *Load from File* menu options handle location and retrieval of stored knowledge. In the previous section, the *RegisterConversation* object was stored in ARAMS. Below is the process for locating and retrieving that same conversation. Figure 22 (specifically steps 4-6 of that figure) captures this process generically.

1) Because the current version of agentTool uses pointers, a user must create a generic conversation and select it.

2) Select *Restore Object* option, which will either create a new *connectionAgent* or use the existing one if it is still active.

3) The *LoadGUI* appears (Figure 31). Currently the user must select the knowledge type he intends to load, which is conversations in this case. This initiates the *CAList* side of a conversation with the ARAMS Administrator agent by passing the selected *objecttype* to it.

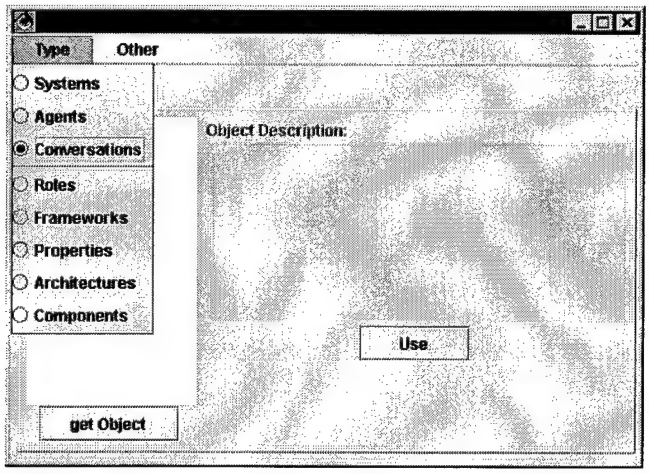


Figure 31: LoadGUI (selecting a knowledge category)

4) The *CList* conversation side send a list request message to the Administrator agent, which initiates its half of the conversation as *KBList*.

5) *KBList* calls its parent agent's *list* method which extracts a Collection (discussed in Chapter 4) of keys for the selected knowledge type. For this example, a list of conversation keys/identifiers is produced.

6) The key collection is returned to the *connectionAgent,* which causes them to be displayed on the *LoadGUI.*

7) The user selects a candidate from *LoadGUI*'s listing and then clicks the *GetObject* button. This starts the *CALoadObject* conversation side by passing it an *ATKB_Object* as done in the Store example above).

8) The ARAMS administrator launches its corresponding side of the conversation, called *KBLoadObject*. The key is checked for a match (there should be one).

9) If there is a match, the parent agent's *Load* method is run, which retrieves the MAML string corresponding to the key.

10) The MAML code is returned to the connection and the MAML description field is displayed in *LoadGUI* (Figure 32). If the described object is what the user desired, then he clicks the *Use* button, which replaces the object selected in agentTool with the loaded object by passing the MAML string into the object's IntegrateKBInfo method. This method simply reconstructs the object from the MAML code.

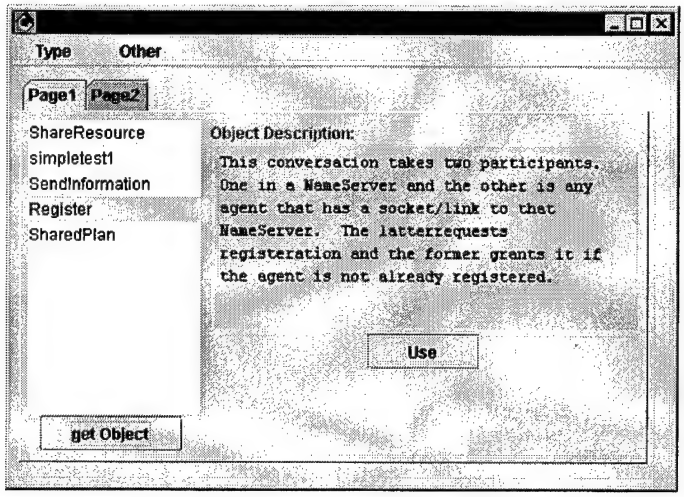


Figure 32: LoadGUI Displaying a List and Selected Conversation object

This covers the restore function of agentTool and how it interfaces with ARAMS to provides knowledge persistence and reuse.

5.4 Summary

Several load and store tests similar to the given example were performed using ARAMS and agentTool in distributed environment. These tests were successful and fast in every instance. However, delete and update functions were not tested due to time constraints. Doing so is considered future work. Additional areas of future work are suggested in Chapter 6.

VI. Results

This chapter focuses on the Knowledge Base Development Methodology (KBDM) and its resulting ARAMS Knowledge Base. Strengths and weaknesses of these two aspects of this effort are presented in Section 6.1. Suggestions for possible applications and future work are then suggested in Section 6.2. Finally, Section 6.3 provides a conclusion.

6.1 Strengths and Weaknesses

Chapter 1 summarized the goal of this effort as: first, developing a methodology for designing an agent knowledge base to support agent synthesis; second, producing a prototype knowledge base by following that methodology; and third, testing the prototype. That testing required evaluation in the areas of: interoperability with agentTool, persistence and reusability of content, extensibility, and reliability. Both the methodology and prototype are critiqued here.

This effort required development of a methodology for producing a workable knowledge base. This was accomplished. The result, KBDM, is both logical in its progression, and clear in its goals (Chapter 3). Of the five key products of KBDM, all can be traced to initial requirements. In part because of the key products, KBDM was straightforward and completely successful in what it was established to do: formalize development of a knowledge base prototype. Though there are no obvious limitations to KBDM, more extensive domain research could have produced a more rigorous domain model. The domain model product of KBDM, however, is sufficient as a baseline and is even permitted to evolve by iterative application of KBDM.

The final product of KBDM was the prototype ARAMS knowledge base. ARAMS *interoperated* smoothly with agentTool due to the MAML-Java object parsing functions. All agentTool knowledge modeled as Java objects was able to be stored and retrieved for *reuse* using

MAML and ARAMS. While stored, this knowledge was kept in secondary storage, which *persisted* even when the ARAMS management threads were terminated and later restarted. This was proven by intentional termination of the ARAMS processes, which otherwise ran *reliably* even over extended periods of time. Unlike reliability, persistence, and interoperability with agentTool, extensibility of MAML-ARAMS was not easily proven. MAML did succeed in providing flexibility to capture changes to the agentTool domain model, which are identified using a *version* tag. However, that flexibility required programmer intervention in updating the object-MAML parse methods. The following section indicates one means of removing this overhead for achieving extensibility, XMI.

By accomplishing this short analysis, we have implicitly discussed several of the strengths and weaknesses of KBDM and ARAMS. Another issue that may, in effect, be either a weakness or strength is the flexibility of applying KBDM in other domains. This idea, in part, is addressed in the next section.

6.2 Applications and Future Work

There are several activities that may follow-on from this effort, ranging from formalization of expansion of the domain itself, to powerful new uses and extensions of MAML. In the middle is expansion of agentTool to use untested abilities of the prototype (such as filtering knowledge by association). Formalization of domain model organization would permit automation of adding domain knowledge to agentTool and ARAMS through known software engineering design processes. Whether that new information is added formally or in traditional fashion, there are several domain dimensions that could be targeted for expansion. Here are three:

1) **Conversations**: Conversations in the current version of agentTool are modeled strictly as dialogues (for two party communications). In reality, a conversation does not have to be a dialogue, because it could occur between as a multicast or broadcast to multiple

parties. To design agent systems taking advantages of the full range of communication possibilities, the conversation dimension of the agent domain needs to be reworked. This would likely incur changes in the MAML grammar and DTDs.

2) **Cognitive Features**: In KBDM domain analysis, properties and capabilities of agents and their components were discussed. Though these are helpful for developing an agent that meets specific functional needs, there may be a need to identify agents that have certain cognitive features as well. For example: an agent may be most suited for a specific environment type (static, dynamic, simulated, real-world, etc). Or maybe the user needs an agent that can handle certain environmental events better than others (unpredictable, asynchronous, concurrent, timed). Capturing these cognitive features in the agent domain model would enrich the property dimension and indirectly increase the utility of the relationships entities that function to provide users options based on requirements.

3) **Agent Theories**: If the process of adding broad new classes of knowledge is to be achieved, agent theories need to be given due consideration. Though Wooldridge and Jenning's work on agent theories was given only a cursory examination in Chapter 2, their efforts provide an ideal starting point. In short, they introduce agent theories as collections of logical principles/concepts that allow for specification of agents. Since the products of KBDM (e.g. domain model, knowledge base, etc) directly support specification of agents at multiple levels, KBDM might be formalized so that those products support either specific theories or a meta-theory that encompasses multiple theory possibilities.

MAML's XML ties make it another likely target for future work. For instance, the eXtensible Styling Language (XSL) may be applied to MAML (W3C 1998). This application may be in the form of providing scripts for automated translation from one version of the MAML to another. The reader may recall that a key feature of MAML was its ability to capture different versions of the agentTool object model, which could be then mapped to the current version as needed. Doing so, however, involved programmer intervention. XSL would automate the process by formalizing all necessary mappings in a set of DTDs for a Multi-Agent Styling Language (MASL). An alternative to using XSL that can produce similar results is) the Document Object Model (DOM) (W3C 1998), and XMI. DOM standardizes the structure of XML documents (objects in the case of MAML). By doing this, direct mapping from one XML variant to another, such as MAML to HTML for viewing in a browser, is facilitated. Besides aiding translation, DOM also assists direct creation of XML documents by growing them as a tree structure rather

than by adding various tags to a flat text file (McGrath 1998). Some preliminary work with the DOM was accomplished in this effort in the form of a utility used for viewing MAML-represented knowledge entities as an object tree. Figure 33 below shows the output of that utility for one agent system captured in MAML. Other XML-related technologies are also forthcoming. One is schematic XML, which allows for user-defined data types and other useful extensions. Another, the extensible XML Metadata Interchange standard (XMI), provides a complete meta-meta structure format for XML documents (and a meta-structure for DTDs) (OMG 1998). In short, this allows for auto-generation and exchange of different XML DTDs between proprietary tools and versions of tools without regard to the source. Taking advantage of XMI for MAML would allow agentTool MAML agents to be used by other agent utilities and for those tool products to be used by agentTool. Implicit synergy between currently disparate research efforts would be the results of doing this.

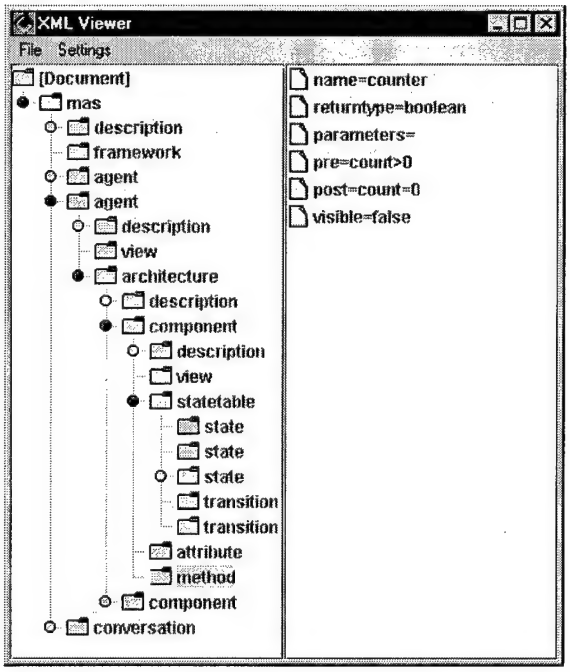


Figure 33: DOM-viewer Showing an Agent System Represented by MAML

6.3 Conclusion

In Chapter 1, the motivating force of this effort was introduced as being a military need that is arising from quickly-evolving technology both here and abroad, in concert with ever-more strained manpower for operating and otherwise interfacing with this technology. Agents can act as a source of virtual manpower to assist the military. However, agents require manpower for their own development, deployment, and maintenance. This "catch-22" can be circumvented through the application of tools such as agentTool that speed effective, tailored development of agent forces. The development of the ARAMS as a knowledge base for agentTool directly supports this goal. ARAMS maintains agent development know-how in persistent storage for on-demand implementation. Population of the ARAMS is, for the most part, done through agentTool, allowing security, robustness, and consistency to be centrally controlled.

KBDM (or a close variant) is envisioned to be applicable in other areas where knowledge reuse and persistence is a requirement. Additionally, agentTool, KBDM, and ARAMS can, at a minimum, provide templates for creation of tools to serve the US Air Force in the coming century. Finally, agentTool, via MAML, may become one application in a virtual suite of agent development and test tools that are brought together via a shared meta-domain model.

APPENDIX A: Domain Analysis Requirements Document

Domain Analysis Requirement Document 10 Jan 2000

Definition of Domain: *A bounded sphere of activity, interest, or function (or knowledge).*

Name of Target Domain: *Agent*

Definition of Target Domain: *A collection of knowledge for multiple dimensions(aspects) of agent; including: concepts, specifications, designs, and implementations of agent systems, agents, and agent constituent components that together completely capture agent information at every level of representation (abstraction) and stage of development*

Relation to SW Engineering Process: *Domain model will be used by agentTool application in the form of a reliable, extensible knowledge base system containing reusable and persistent agent domain knowledge. AgentTool uses the MaSE methodology in applying that knowledge (Wood 2000).*

MaSE Stage	New Knowledge Elements Used
Capturing Goals	*Goals, Use Cases, and Goal Hierarchy*
Transforming To Roles	*Use Cases & Roles*
Applying Use Cases	*Tasks*
Creating Agents	*Sequence Diagrams*
Creating Conversations & Assembling Agents	*Conversations, Components, Connectors, and Architectures*
System Deployment	*Implementations*

Requirements:

- Domain model and other products must support MaSE knowledge needs.
- Products must be implemented in Java in order to interface with agentTool.

Assumptions:

- Domain analysis may extend beyond the MaSE areas so as to permit operation of alternative development methodologies in the future.
- Domain model should be capable of evolving as a byproduct of future domain analysis iterations.
- Agent theories and language will not be captured in this analysis iteration
- There are specification, design, and implementation levels and possibly a conceptual level of agent creation. Elements in the MaSE table reflect objects at all design levels.

Problem:

- Evolving domain model may cause version incompatibilities. Resolve this.

Recommendations:

- Perform domain analysis in: the agent-entity and agent-system sub-domains
- Capture meta-knowledge for mapping design-level knowledge to implementation-level knowledge

Collected Knowledge: Here is a short index to the research summarized in Section 2.3.

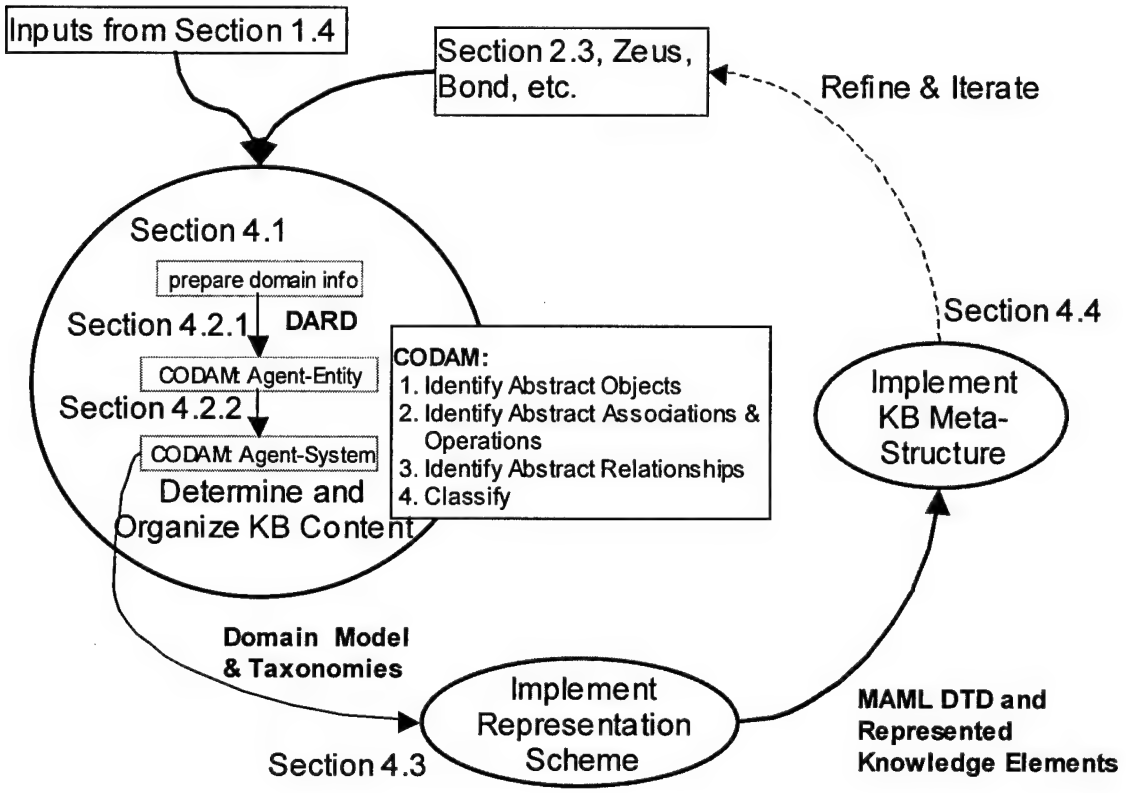
- Franklin and Graesser's classification of agents by properties and components.
- University of Michigan's delineation of agent architectures and distinguishing of them by their properties and components
- McGill University's UAA, an attempt to create an abstract agent architecture with universally usable components and implementation framework
- University of Cincinnati and Stanford's JAFMAS and JATLite communication frameworks and communication specifications.
- Elizabeth Kendall's analysis and specification of roles, role models, and behaviorally modeled RMIT agent architecture.
- FIPA's various specifications for standardizing and defining agent relationships, architecture, and constitution.

Existing Systems:

- Zeus Agent Toolkit (Ndumu 1999)
- IBM Aglets (IBM 1999)
- AgentBuilder (Reticular 1999)
- BOND (Boloni 1999)

Knowledge Engineer: *Marc J Raphael*

APPENDIX B: Visual Summary of KBDM for Effort



APPENDIX C: Key MAML DTDs

(These are accurate as of 12Dec99)

System DTD

```
<?xml encoding="US-ASCII"?>
<!-- The Document Type Definition of atsystem.xml-->
<!ENTITY % agent-grammar SYSTEM "atagent.dtd">
<!ENTITY % conversation-grammar SYSTEM "atconversation1.dtd">

<!ELEMENT mas (description?, framework?, agent*, conversation*)>
<!ATTLIST mas version CDATA #REQUIRED name CDATA #REQUIRED>
<!ELEMENT description (#PCDATA)>
<!ELEMENT framework EMPTY>
<!ATTLIST framework name CDATA #REQUIRED>
```

Agent DTD

```
<?xml encoding="US-ASCII"?>
<!-- The Document Type Definition of atagent.xml-->
<!--ENTITY % architecture-grammar SYSTEM "atarchitecture.dtd"-->

<!ELEMENT agent (description,view?,property*)>
<!ATTLIST agent
  version CDATA #REQUIRED
  name    CDATA #REQUIRED>
  <!ELEMENT description (#PCDATA)>
  <!ELEMENT view EMPTY>
  <!ATTLIST view
    x CDATA #IMPLIED
    y CDATA #IMPLIED
    w CDATA #IMPLIED
    h CDATA #IMPLIED>
  <!ELEMENT property    (#PCDATA)>
```

Architecture DTD

```
<?xml encoding="US-ASCII"?>
<!-- The Document Type Definition of atarchitecture.xml-->
<!ENTITY % component-grammar SYSTEM "atcomponent.dtd">

<!ELEMENT architecture (description?, component*)>
<!ATTLIST architecture
  name CDATA #REQUIRED
  category(REACTIVE|DELIBERATIVE|COMPOSITE|BDI|OTHER) #IMPLIED>
  <!ELEMENT description (#PCDATA)>
```

Role DTD

```
<?XML ENCODING="US-ASCII"?>
<!-- THE DOCUMENT TYPE DEFINITION OF ATROLE.XML-->
  <!ELEMENT ROLE ()>
  <!ATTLIST ROLE
```

```
NAME CDATA #REQUIRED
```

Conversation DTD

```
<?xml encoding="US-ASCII"?>
<!-- The Document Type Definition of atconversation.xml-->
<!ELEMENT conversation   (participant*)>
<!ATTLIST conversation
  name    CDATA #REQUIRED
  version CDATA #REQUIRED>
  <!ELEMENT participant(description,statetable?)>
  <!ATTLIST participant
    name (initiator|responder|multicastparticipant) "initiator">
    <!ELEMENT description (#PCDATA)>
    <!ELEMENT statetable(state*, transition*)>
      <!ELEMENT state   (view)>
      <!ATTLIST state
        name   CDATA #IMPLIED
        action CDATA #IMPLIED>
        <!ELEMENT view EMPTY>
        <!ATTLIST view
          x CDATA #IMPLIED
          y CDATA #IMPLIED
          w CDATA #IMPLIED
          h CDATA #IMPLIED>
      <!ELEMENT transition EMPTY>
      <!ATTLIST transition
        rMessage CDATA #IMPLIED
        tMessage CDATA #IMPLIED
        guard    CDATA #IMPLIED
        cstate   CDATA #IMPLIED
        nstate   CDATA #IMPLIED
        action   CDATA #IMPLIED
```

Component DTD

```
<?xml encoding="US-ASCII"?>
<!-- The Document Type Definition of atcomponent.xml-->
<!ENTITY % architecture-grammar SYSTEM "atarchitecture.dtd">

  <!ELEMENT component (description?, view?, architecture?, statetable?,
attribute*, method*)>
  <!ATTLIST component
    name CDATA #REQUIRED
    <!ELEMENT view EMPTY>
    <!ATTLIST view
      x CDATA #IMPLIED
      y CDATA #IMPLIED
      w CDATA #IMPLIED
      h CDATA #IMPLIED>
    <!ELEMENT description (#PCDATA)>
    <!ELEMENT statetable(state*, transition*)>
      <!ELEMENT state   (view)>
      <!ATTLIST state
```

```
    name   CDATA #IMPLIED
    action CDATA #IMPLIED>
    <!ELEMENT view EMPTY>
    <!ATTLIST view
        x CDATA #IMPLIED
        y CDATA #IMPLIED
        w CDATA #IMPLIED
        h CDATA #IMPLIED>
    <!ELEMENT transition EMPTY>
    <!ATTLIST transition
      rMessage CDATA #IMPLIED
    tMessage CDATA #IMPLIED
    guard    CDATA #IMPLIED
    cstate   CDATA #IMPLIED
    nstate   CDATA #IMPLIED
    action   CDATA #IMPLIED
<!ELEMENT method EMPTY>
<!ATTLIST method
  rMessage CDATA #IMPLIED
  tMessage CDATA #IMPLIED
  guard    CDATA #IMPLIED
  cstate   CDATA #IMPLIED
  nstate   CDATA #IMPLIED
  action   CDATA #IMPLIED>
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
  name         CDATA #IMPLIED
  type         CDATA #IMPLIED
  runtimedefined    CDATA #IMPLIED
  userdefined       CDATA #IMPLIED
  collectiontype    (SET|SEQUENCE|BAG) #IMPLIED>
```

Bibliography

Agentsoft. (1998). "Agentsoft and Automation Technology White Paper." .

Arango, G., and Prieto-Diaz. (1991). "Domain analysis concepts and research directions." *Domain Analysis and Software Systems Modeling*, A. a. Prieto-Diaz, ed., IEEE Computer Society Press, Los Alamitos, CA.

Arriola, G. a. o. (1994). "A Survey of Cognitive and Agent Architectures." *for EECS 547, Cognitive Architectures*, University of Michigan.

Belgrave, M. (1995). "The Unified Agent Architecture: A White Paper." , McGill University, Montreal, Canada.

Bjork, R. (1998a). "CS352: Object-Oriented Databases." , Gordon College, Wedham, MA.

Bjork, R. (1998b). "CS352: The Relational Data Model." , Gordon College, Wenham, MA.

Boloni, L. (1999). "BOND." , Purdue University, Layfayette, IN.

Brodie, M. a. F. M. (1989). "Database Management: A Survey." Foundations of Knowledge Base Management, J. W. a. C. T. Schmidt, ed., Springer-Verlag, Harrisburg, VA, 205-235.

English, J. (1998). "The Brighton University Resource Kit for Students." , University of Brighton, United Kingdom.

FIPA. (1999a). "Architectural Principles Baseline." *FIPA Spec 0-1999*, Foundation for Intelligent Physical Agents.

FIPA. (1999b). "FIPA Architectural Overview." *FIPA 9710*.

Franklin, S. a. A. G. "Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents." *Third International Workshop on Agent Theories*.

Frost, R. a. C. "Design for Manufacturability via Agent Interaction." *ASME Computers in Engineering Conference*.

Galdarin, G. a. P. V. (1989). *Relational Databases and Knowledge Bases*, Addison-Wesley, Reading, MA.

Gallaire, H. a. J.-M. N. (1989). "How to Look at Deductive Databases." Foundations of Knowledge Base Management, J. W. a. C. T. Schmidt, ed., Springer-Verlag, Harrisburg, VA, 119-126.

Gonzalez, A. J. (1993). *Engineering of Knowledge-based Systems*, Prentice Hall, Englewood Cliffs, NJ.

Guha, R. a. D. L. (1994). "Enabling Agents to Work Together." *Communications of the ACM (CACM)*, Volume 37, 126-142.

Hamner, D. (1999). "Use a RandomAccessFile to building a low-level database." JavaWorld.

IBM. (1999). "Aglets Software Development Kit." , IBM.

Kendall, E. (1998a). "Role Modeling for Agent System Analysis, Design, and Implementation." , Royal Melbourne Institute of Technology, Melbourne, Australia.

Kendall, K., Pathak, and Suresh. "A Java Application Framework for Agent Based Systems." *ACM Computing Surveys Symposium on Application Frameworks.*

Lacey, T. (2000). "A Formal Methodology and Technique for Verifying Conversations in a Closed Multi-agent System," , Air Force Institute of Technology, Dayton.

Lukose, D. a. R. K. (1996). "Knowledge Engineering." , University of Calgary, Calgary, Canada.

McGrath, S. (1998). *XML by Example: Building e-Commerce Applications*, Prentice-Hall, Upper Saddle River.

Minsky, M. (1975). "A Framework for Representing Knowledge." The Psychology of Computer Vision, P. H. Winston, ed., McGraw-Hill, New York, NY.

Ndumu, O. a. (1999). "Zeus Toolkit." , British Telecommunications, United Kingdom.

Nixon, B. e. a. (1989). "Design of a Compiler for a Semantic Data Model." Foundations of Knowledge Base Management, J. W. a. C. T. Schmidt, ed., Springer-Verlag, Harrisburg, VA, 293-343.

Nwana, H. (1996). "Software Agents: An Overview." *Knowledge Engineering Review*, 11(3), 205-244.

Olsen, J. D. a. R. E. K. "Conceptual Knowledge Markup Language, an XML Application." *XML Developer's Day*, Montreal, Canada.

OMG, O. M. G. (1998). "XML Metadata Interchange (XMI)." , 20 October 1998.

Pedersen, K. (1989a). "Well-Structured Knowledge Bases Part I." Validating and Verifying Knowledge-based Systems, U. Gupta, ed., IEEE Computer Press Society, Los Alamitos, CA, 365-276.

Pedersen, K. (1989b). "Well-Structured Knowledge Bases Part II." Validating and Verifying Knowledge-based Systems, U. Gupta, ed., IEEE Computer Press Society, Los Alamitos, CA, 377-380.

Reticular. (1999). "AgentBuilder." , Reticular Systems, San Diego.

Robinson, D. J. (2000). "A Component Based Approach to Agent Specification," , Air Force Institute of Technology, Dayton.

Schmidt, J. W. a. C. T. (1989). "Preface." Foundations of Knowledge Base Management, J. W. a. C. T. Schmidt, ed., Springer-Verlag, Harrisburg, VA, VII-IX.

Sernadas, A. a. C. S. (1989). "Abstraction and Inference Mechanisms for Knowledge Representation." Foundations of Knowledge Base Management, J. W. a. C. T. Schmidt, ed., Springer-Verlag, Harrisburg, VA, 91-111.

Thomas. (1999). "An Agent Generator." , Bits & Pixels.

VandenBerghe, D. (1999). "Re: Intermediate language." .

W3C, W. W. W. C. (1998). "Document Object Model (DOM) Level 1 Specification." , MIT.

Warner, R. M. (1993). "A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System," Masters, AFIT, Dayton.

Webster. (1986). *Collegiate Dictionary*, Merriam-Webster, Springfield, MA.

Williams, C. (1990). "Expert Systems, Knowledge Engineering, and AI Tools - An Overview." Expert Systems: A Software Methodology for Modern Applications, E. Nahouraii, ed., IEEE Computer Press Society, Los Alamitos, CA.

Wood, M. (2000). "Multiagent Systems Engineering: A Methodology For Analysis and Design of MultiAgent Systems," , Air Force Institute of Technology, WPAFB.

Wooldridge, M. a. N. J. (1995). "Intelligent Agents: Theory and Practice." *Knowledge Engineering*.

Vita

Marc Joseph Raphael was born in the small town upstate New York city of Norwich in May 1971. He graduated from Norwich High School in 1989. That same year he won an Air Force scholarship to Brigham Young University. He interrupted that secondary education in 1990 in order to serve in Chile as a volunteer missionary. Upon returning to his studies in 1992, Marc completed a Bachelor's degree in Electrical and Computer Engineering at BYU (1995). In December 1995, he was commissioned in the US Air Force and left for a tour at NAIC, Wright-Patterson AFB, Ohio. While at NAIC he earned a Meritorious Service Medal for his support in multiple vital computer-related programs. When the NAIC tour ended, 1Lt Raphael entered AFIT to complete a Master's degree in Computer Engineering. Upon completion of that degree in March 2000 Captain Raphael was assigned to serve at the Space and Missile Systems Center at LA AFB, CA.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE

KNOWLEDGE BASE SUPPORT FOR DESIGN AND SYNTHESIS OF MULTI-AGENT SYSTEMS

5. FUNDING NUMBERS

6. AUTHOR(S)

Marc J. Raphael, Captain, USAF

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/00M-21

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
Attn: Captain Freeman Alex Kilpatrick
801 North Randolph Street
Arlington, VA 22203-1977 Commercial: (703) 696-6565

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Maj Scott A. DeLoach, ENG, DSN:785-3636, ext. 4622

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

ABSTRACT (*Maximum 200 Words*)

AgentTool is an AFIT-produced, AFOSR-sponsored multi-agent system (MAS) development tool intended for production of MASs that meet military requirements. This research focuses on enabling MAS design and synthesis tools like agentTool to store, retrieve, and filter persistent, reusable, and reliable agent domain knowledge. This "enabling" is vital if such tools are expected to produce consistent, maintainable, and verifiable agent applications on short timetables. Enabling requires: 1) modeling the agent knowledge domain, 2) designing and employing a persistent knowledge base, and 3) bridging that domain model to the knowledge base with an extensible domain interchange grammar. The achieved interchange grammar, called Multi-Agent Markup Language (MAML), is presented and shown to be capable of representing MAS design knowledge in a concise and easily parsed form that is readily stored and retrieved in the knowledge base. The selected knowledge base, called the Agent Random-Access Meta-Structure (ARAMS), is shown to support MAML and operate in a distributed environment that permits sharing of agent development knowledge between various tools and tool instances. Tests of MAML and ARAMS with agentTool are summarized, and related future work suggested.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Agents, agentTool, Domain Model, UML, XML, KBDM, ARAMS, Java, Knowledge Base, Storage, Data Base, Retrieval	110 (115)
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

SN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102